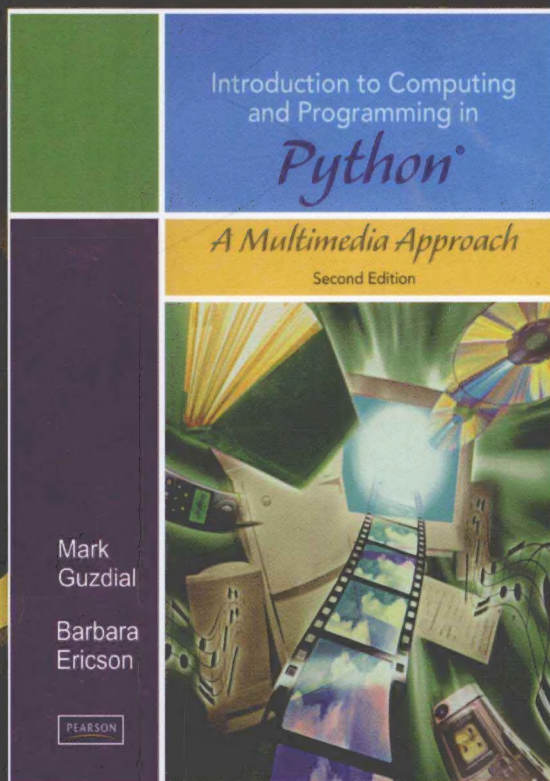


Python计算与编程实践 多媒体方法

(美) Mark Guzdial Barbara Ericson 著 王江平 译

Introduction to Computing and Programming in Python
A Multimedia Approach Second Edition



计 算 机 科 学 丛 书

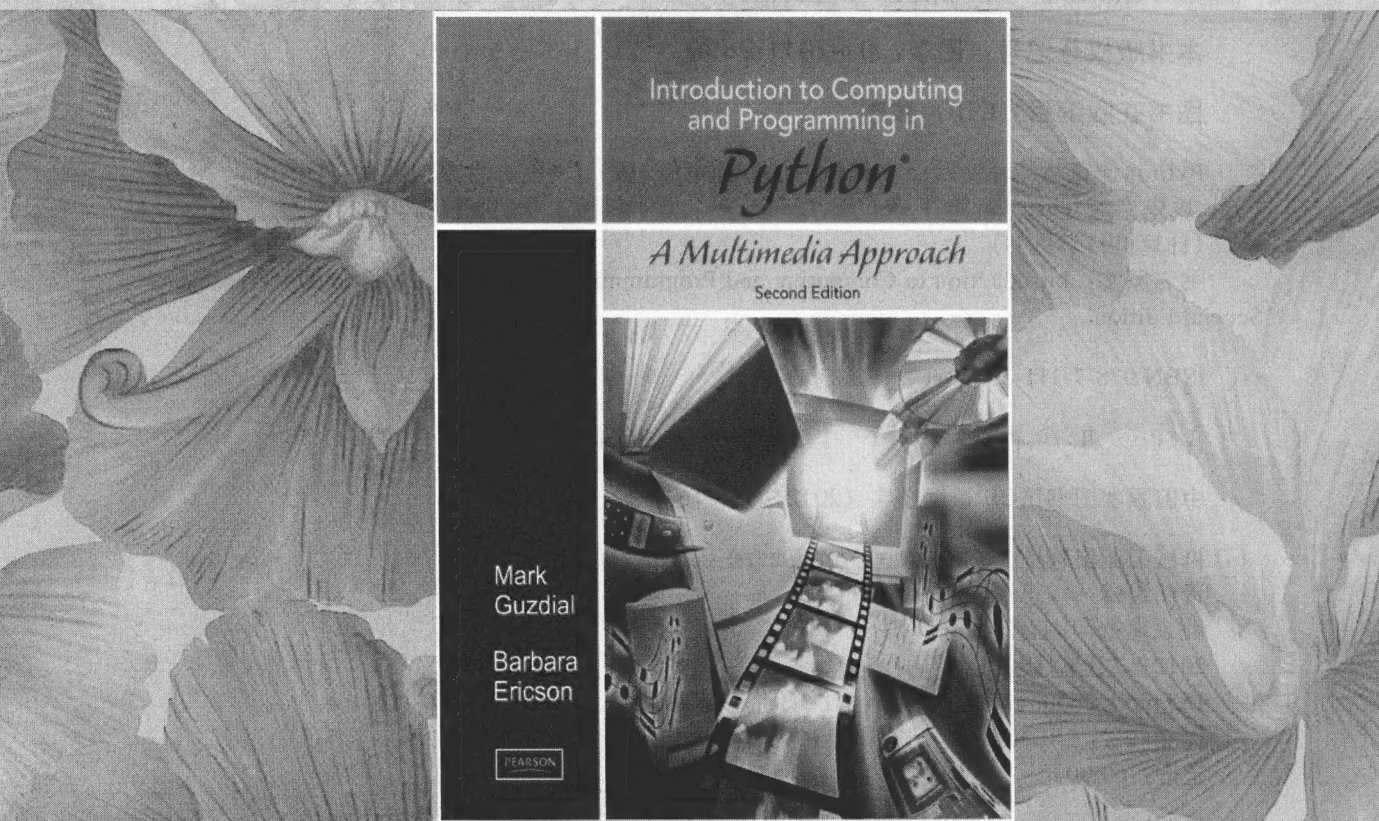
原书第2版

Python计算与编程实践

多媒体方法

(美) Mark Guzdial Barbara Ericson 著 王江平 译

Introduction to Computing and Programming in Python
A Multimedia Approach Second Edition



机械工业出版社
China Machine Press

本书是一本别出心裁的程序设计入门教程,以Python数字多媒体编程为主线,依次讲解了图像、声音、文本和电影的处理,其中穿插介绍了大量的计算机程序设计基础知识。方法独到,示例通俗易懂,条理清晰,将趣味性和实用性融于讲解之中。

本书适合用做计算机专业导论课或非计算机专业编程课程的教材,也可用做软件开发人员学习计算机数字多媒体处理知识和Python语言的专业参考书。

Authorized translation from the English language edition, entitled INTRODUCTION TO COMPUTING AND PROGRAMMING IN PYTHON: A MULTIMEDIA APPROACH, 2E, 9780136060239 by Mark Guzdial, Barbara Ericson, published by Pearson Education, Inc, publishing as Prentice Hall, Copyright © 2010.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and CHINA MACHINE PRESS Copyright © 2012.

本书中文简体字版由Pearson Education(培生教育出版集团)授权机械工业出版社在中华人民共和国境内(不包括中国台湾地区和香港、澳门特别行政区)独家出版发行。未经出版者书面许可,不得以任何方式抄袭、复制或节录本书中的任何部分。

本书封面贴有Pearson Education(培生教育出版集团)激光防伪标签,无标签者不得销售。

封底无防伪标均为盗版

版权所有,侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号:图字:01-2011-2890

图书在版编目(CIP)数据

Python计算与编程实践:多媒体方法(原书第2版)/(美)古兹迪阿尔(Guzdial, M.), (美)埃里克森(Ericson, B.)著;王江平译. —北京:机械工业出版社, 2012.6

(计算机科学丛书)

书名原文: Introduction to Computing and Programming in Python: A Multimedia Approach, Second Edition

ISBN 978-7-111-38738-1

I. P… II. ①古… ②埃… ③王… III. 软件工具—程序设计 IV. TP311.56

中国版本图书馆CIP数据核字(2012)第121641号

机械工业出版社(北京市西城区百万庄大街22号 邮政编码 100037)

责任编辑:盛思源

北京瑞德印刷有限公司印刷

2012年7月第1版第1次印刷

185mm×260mm · 22印张(含1.5印张彩插)

标准书号:ISBN 978-7-111-38738-1

定价:69.00元

凡购本书,如有缺页、倒页、脱页,由本社发行部调换

客服热线:(010) 88378991; 88361066

购书热线:(010) 68326294; 88379649; 68995259

投稿热线:(010) 88379604

读者信箱:hzjsj@hzbook.com

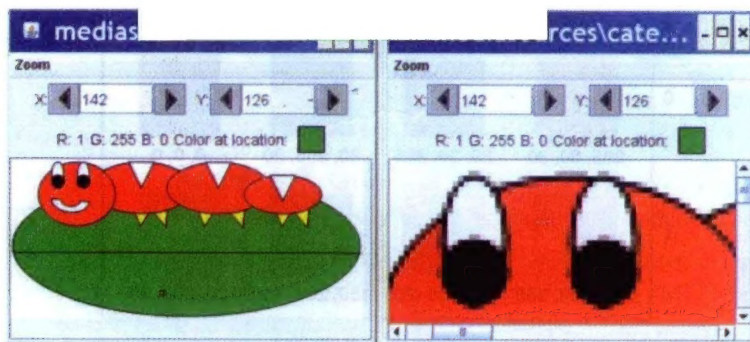


图3.3 显示在JES图片工具中的图像：左侧的显示比例是100%，右侧的显示比例是500%



图3.4 融合红、绿、蓝产生新颜色

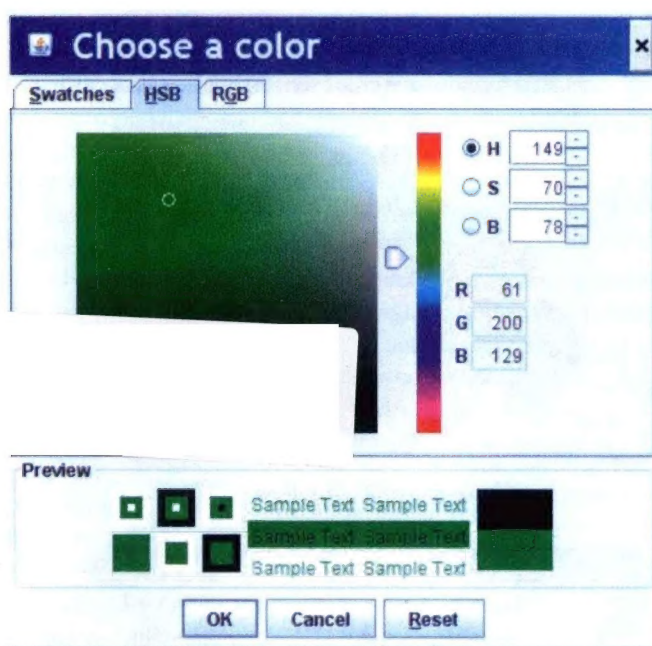


图3.5 基于HSB颜色模型选取颜色



图3.6 此图两端是同样的灰色，但中间的部分对比明显，因此左端看上去比右端更暗









	0	1	2	3
0	 255, 30, 30	 30, 30, 255	 30, 255, 30	 0, 0, 0
1	 255, 150, 150	 150, 150, 255	 150, 255, 150	 200, 200, 200

图3.8 以矩阵表示的RGB三元组

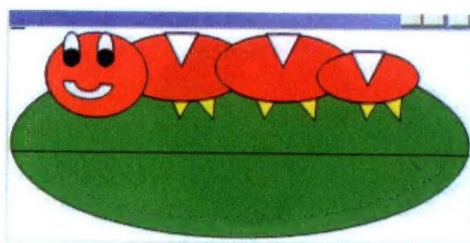


图3.9 使用命令直接修改像素颜色：注意左上角的短小黑线

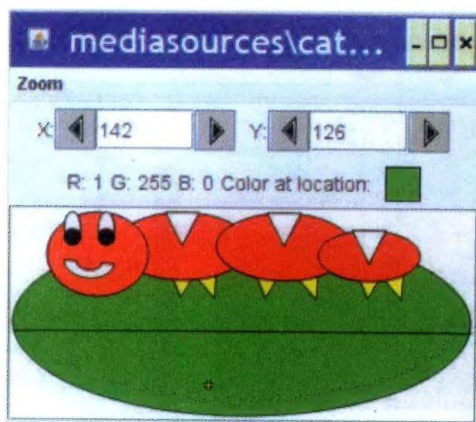


图3.13 在JES图片工具中选择像素

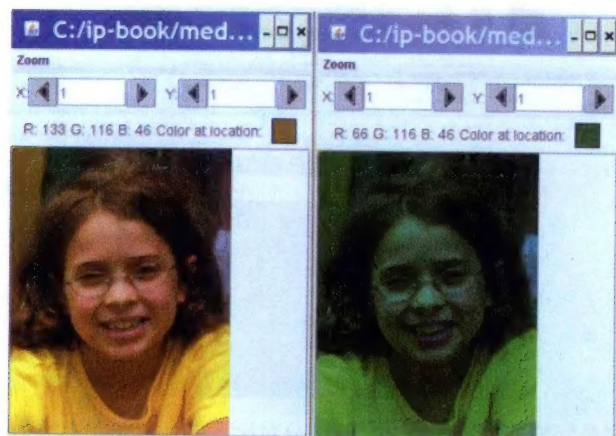


图3.15 原来的图片（左）和减少红色的版本（右）

picture



picture



getPixels()

Pixel, color	Pixel, color	Pixel, color
r=135 g=131 b=105	r=133 g=114 b=46	r=134 g=114 b=45

p

picture



getPixels()

Pixel, color	Pixel, color	Pixel, color
r=135 g=131 b=105	r=133 g=114 b=46	r=134 g=114 b=45

p

value = 135

picture



getPixels()

Pixel, color	Pixel, color	Pixel, color
r=67 g=131 b=105	r=133 g=114 b=46	r=134 g=114 b=45

p

value = 135

picture



getPixels()

Pixel, color	Pixel, color	Pixel, color
r=67 g=131 b=105	r=133 g=114 b=46	r=134 g=114 b=45

p

value = 135

picture



getPixels()

Pixel, color	Pixel, color	Pixel, color
r=67 g=131 b=105	r=133 g=114 b=46	r=134 g=114 b=45

p

value = 133

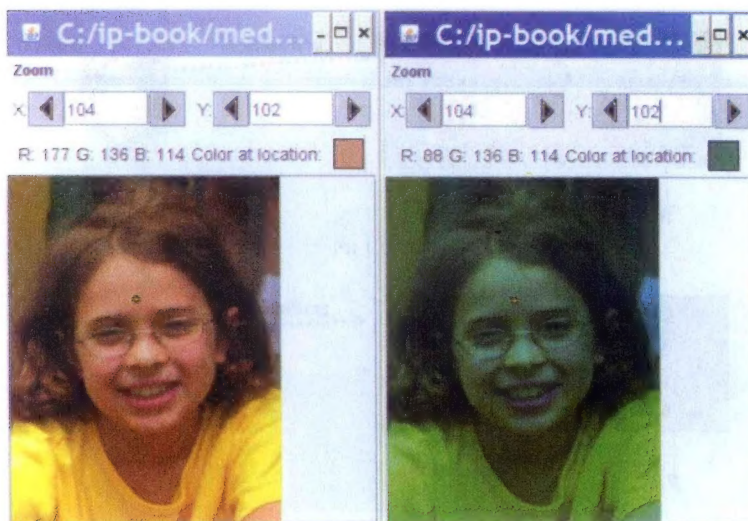
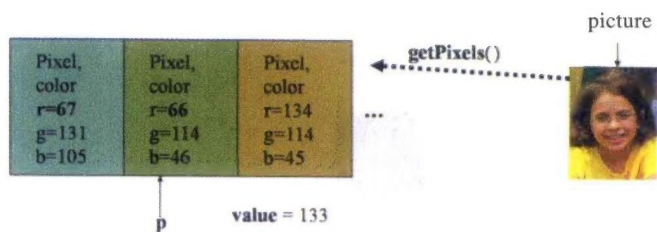


图3.16 使用JES图片工具让自己确信红色已经减少



图3.17 原来的海滩风景（左）和（虚假的）日落时的效果（右）



图3.18 原始图片（左）和更暗的版本（右）



图3.19 底片像

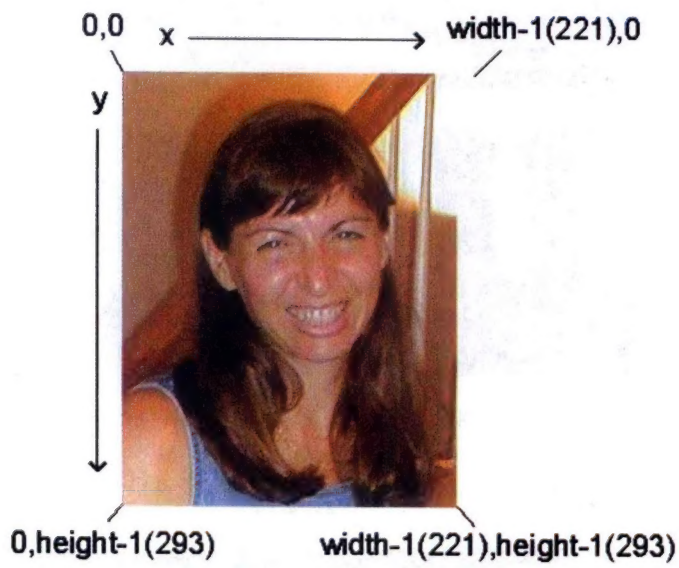


图4.1 图片坐标



图4.3 原图（左）和沿纵向轴镜像后的图片（右）

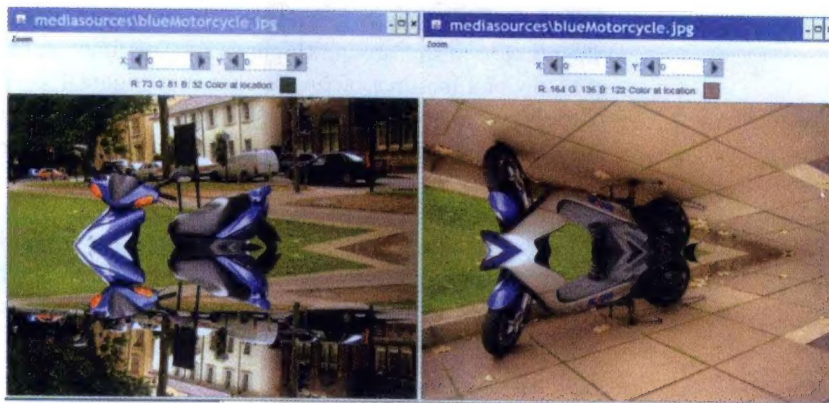


图4.4 横向镜像，从上到下（左）和从下到上（右）



图4.5 摄于雅典古安哥拉遗址的赫菲斯托斯神庙

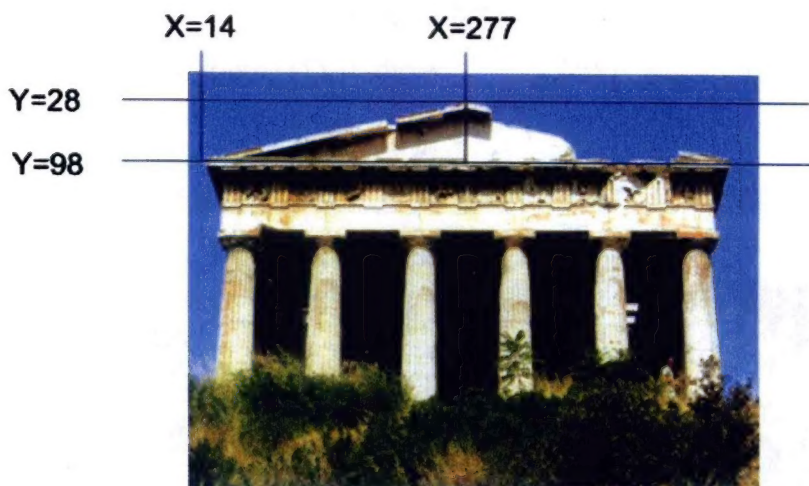


图4.6 需要镜像的坐标位置



图4.7 处理后的神庙



图4.8 将图片复制到画布中 (1)



图4.9 将图片复制到画布中间



图4.11 拼贴图中使用的花朵图片

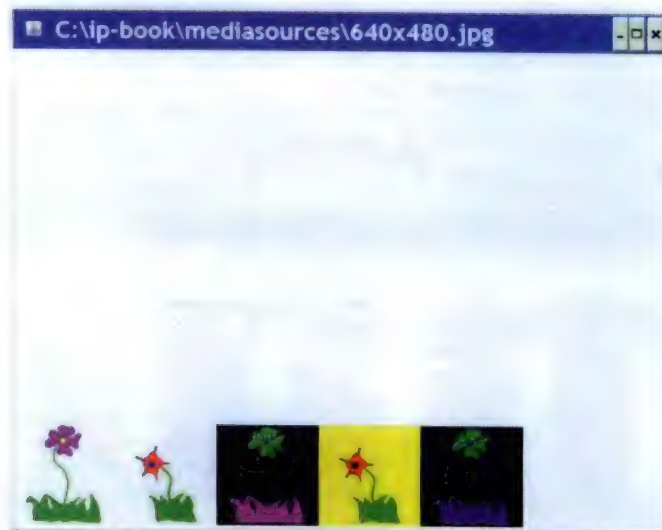


图4.12 花朵拼贴图



图4.13 将图片复制到画布中 (2)



图5.1 把棕色变成红色



图5.2 将矩形区域内的红色加倍

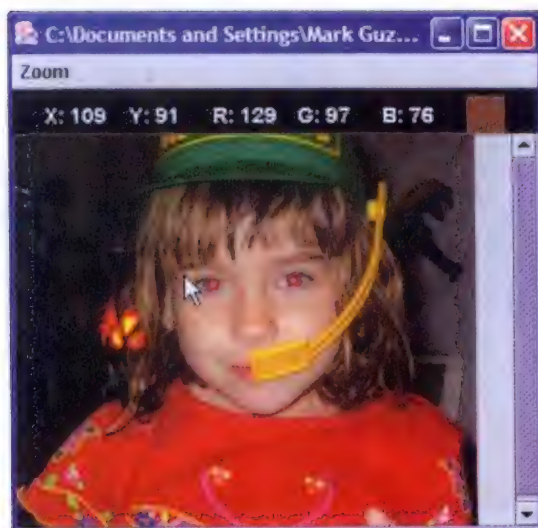


图5.3 找出Jenny的红眼睛区域 (1)

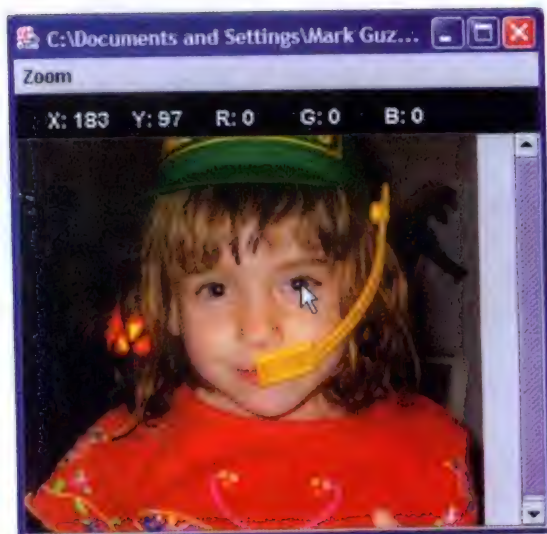


图5.4 找出Jenny的红眼睛区域 (2)



图5.5 原来的风景（左）和使用深褐色调菜谱转换过的（右）



图5.6 把原始图片（左）中的颜色数目减少（右）

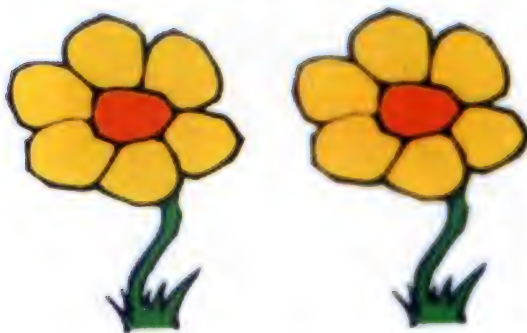


图5.8 将花朵放大（左），然后模糊化以减轻像素化（右）

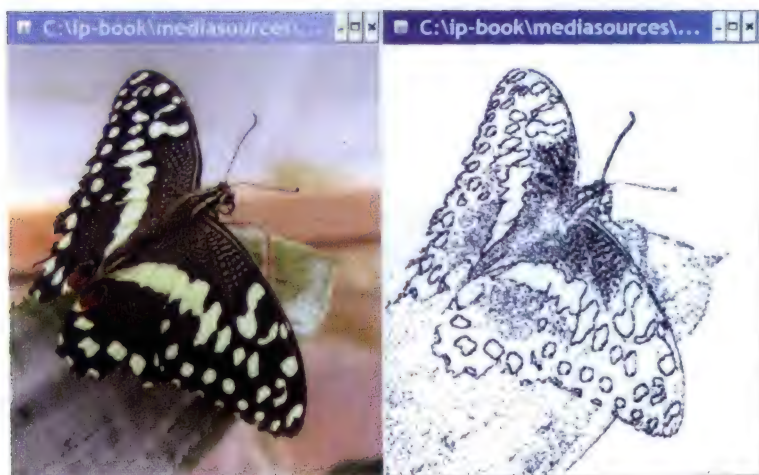


图5.9 转换成“线条画”（右）的蝴蝶（左）



图5.10 融合妈妈和女儿的照片



图5.11 一张小孩（Katie）的照片和上面没有她的背景照片



图5.12 新的背景：月球



图5.13 Katie在月球上

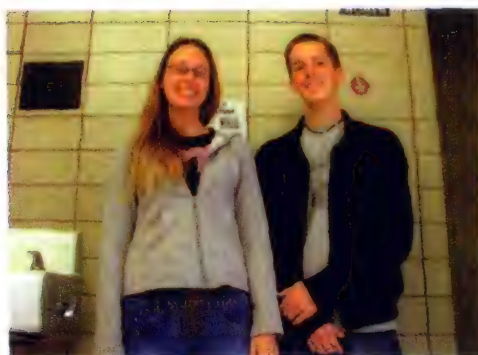


图5.14 两个人站在墙壁前面（左）和墙壁的图片（右）



图5.15 使用背景消减技术用丛林交换墙壁，阈值为50



图5.16 Mark坐在蓝色床单前



图5.17 月球上的Mark



图5.18 丛林中的Mark



图5.19 红色背景前的学生，不用闪光灯（左）和使用闪光灯（右）

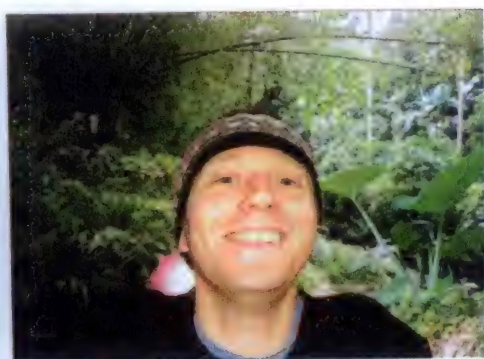
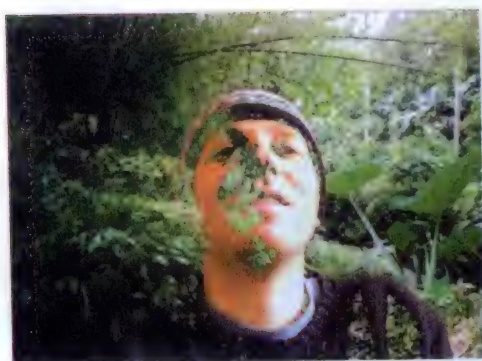


图5.20 使用色键菜谱处理红色背景，不用闪光灯（左）和使用闪光灯（右）



图5.21 原来的（左）和加过线条的（右）Carolina



图5.22 沙滩上漂过来一个盒子



图5.23 画出来的一幅小图片

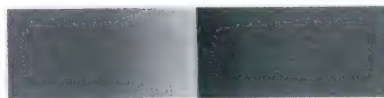


图5.24 程序实现的灰度效果

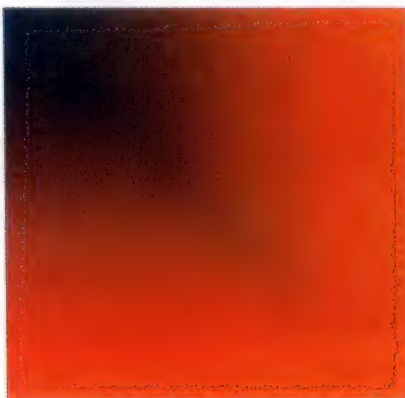


图5.25 嵌套的彩色矩形图像



图5.26 嵌套的空白矩形图像

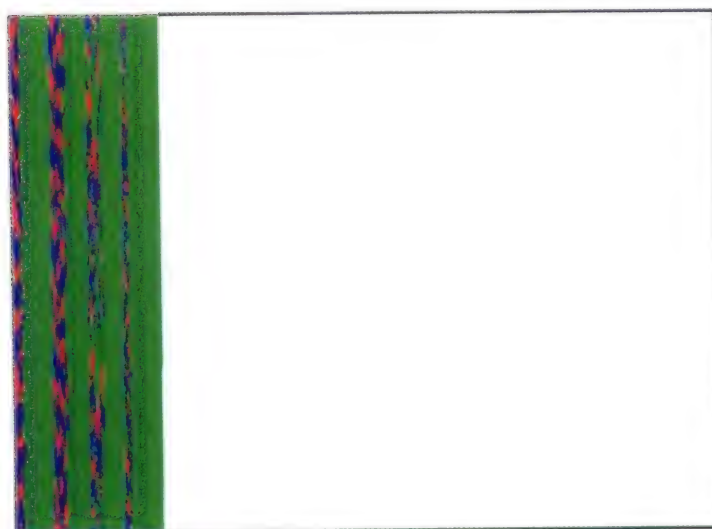


图11.3 声音 “This is a test” 的视觉效果



图11.4 原来的图片（左）和隐藏了信息的图片（右）

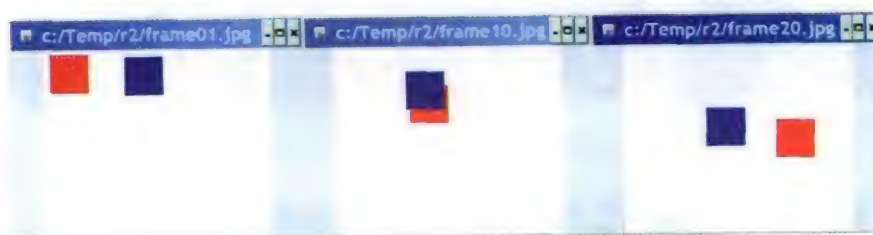


图13.4 同时移动两个方块



图13.5 移动头像电影中的两帧

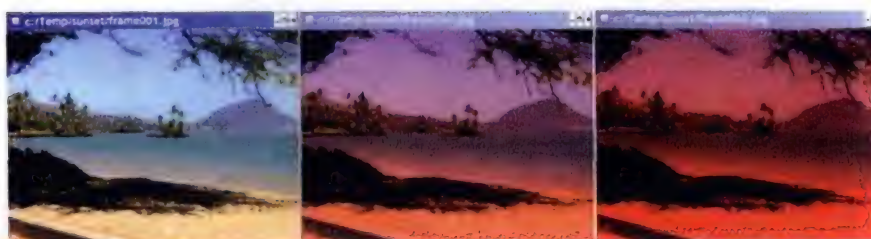


图13-6 缓缓日落电影的多帧画面



图13.7 “缓缓淡出”电影中的几帧画面



图13.9 “妈妈观看Katie跳舞”电影中的几帧画面



图13.10 孩子们在蓝色屏幕前爬行的几帧原始视频画面



图13.11 孩子们在月球上爬行的视频画面

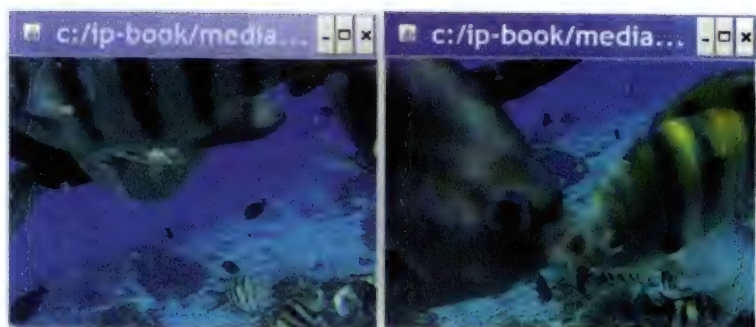


图13.12 蓝色太深的几帧水下电影画面



图13.13 蓝色稍轻的几帧电影画面



图13.14 源电影中的画面，荧光棒在空中画东西



图13.15 目标电影的最后一帧，可以看出荧光棒的轨迹

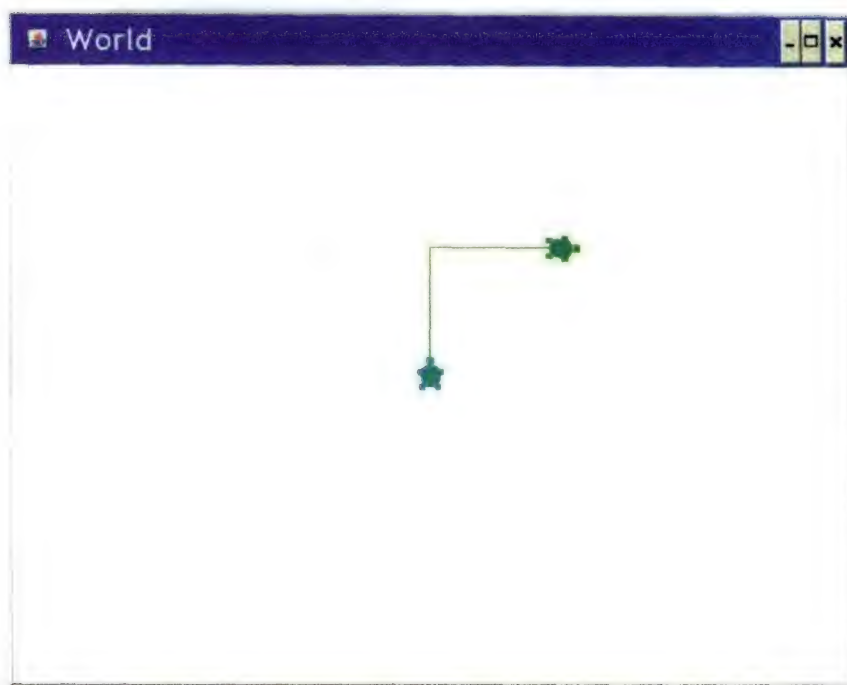


图16.3 让一只小海龟移动并转弯，另一只留在原地

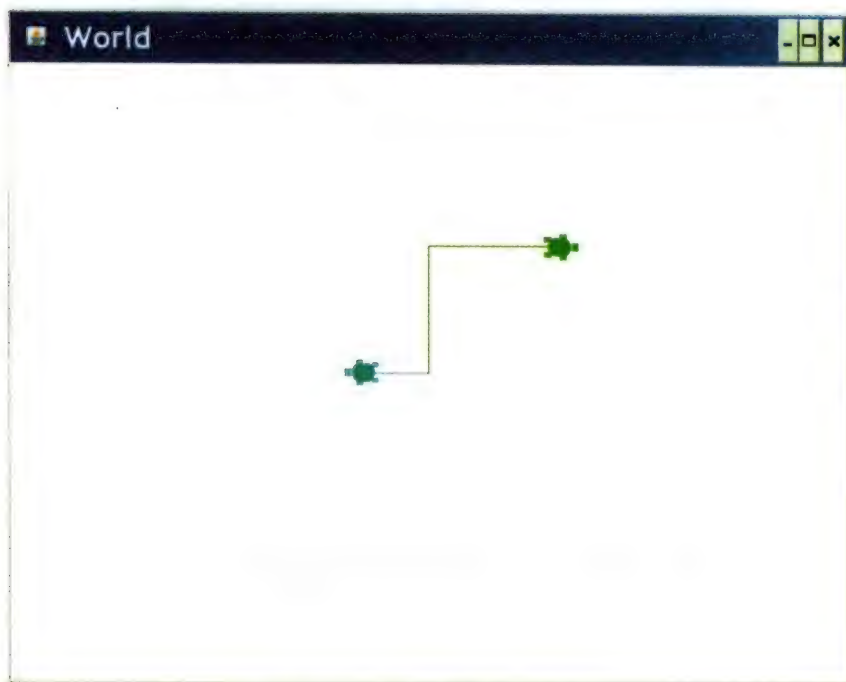


图16.4 第二只小海龟移动之后

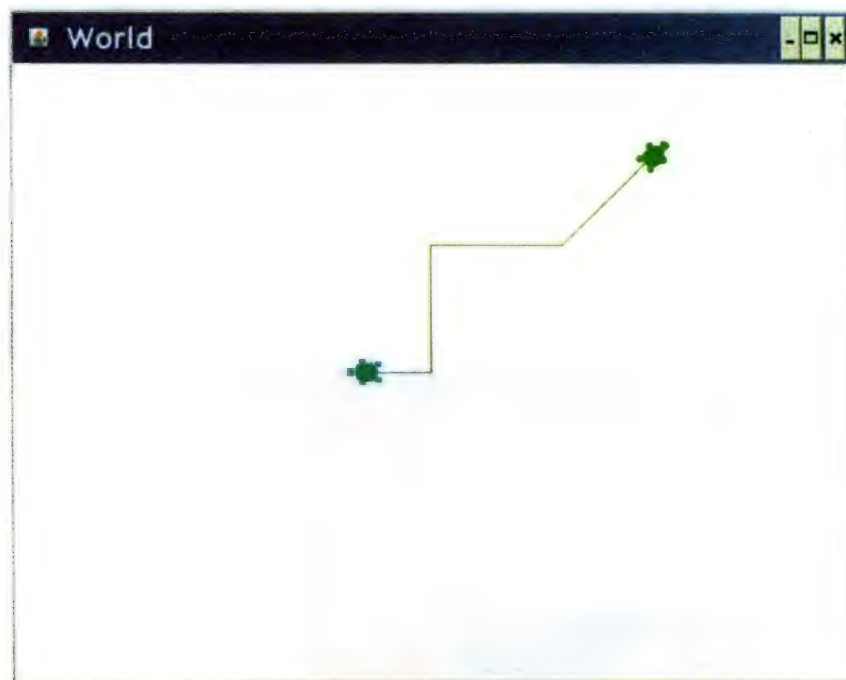


图16.5 旋转指定的度数 (-45°)

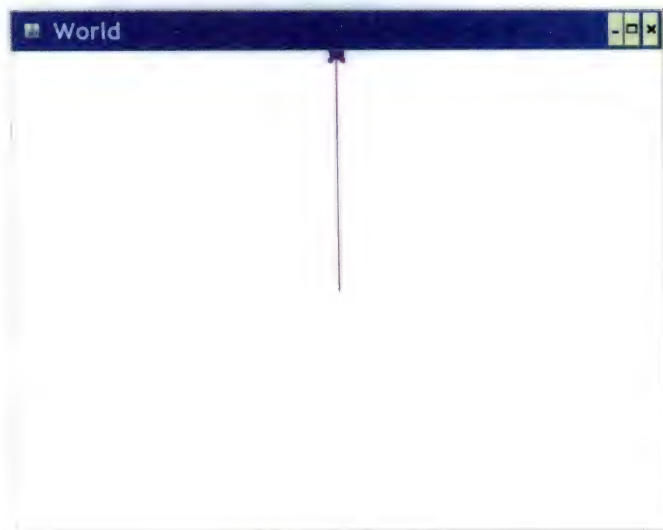


图16.6 停在“世界”边缘的小海龟



图16.7 用小海龟画对角线



图16.8 用小海龟画矩形

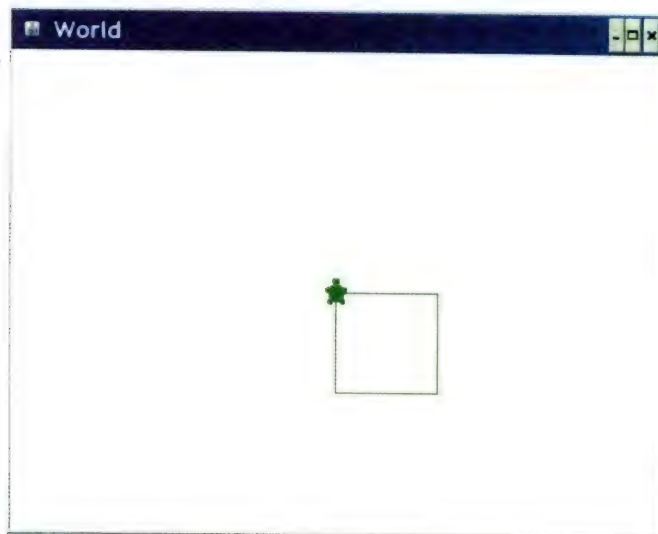


图16.9 使用SmartTurtle画正方形

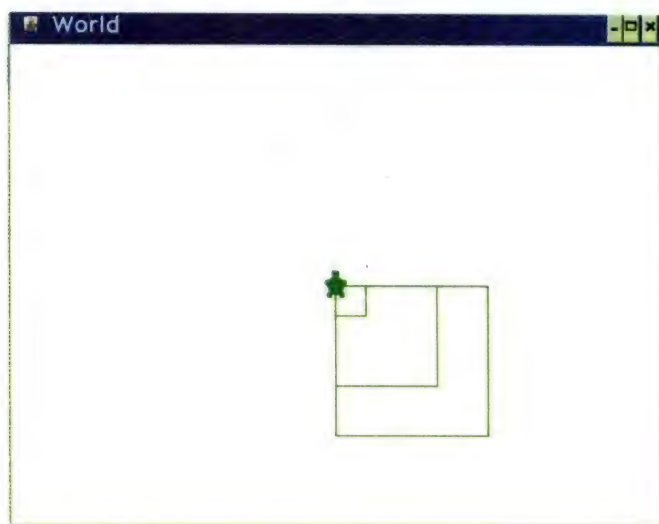


图16.10 画不同尺寸的正方形

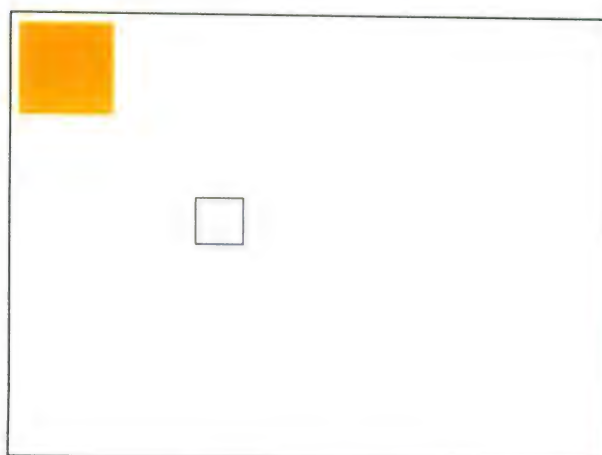


图16.11 矩形方法示例

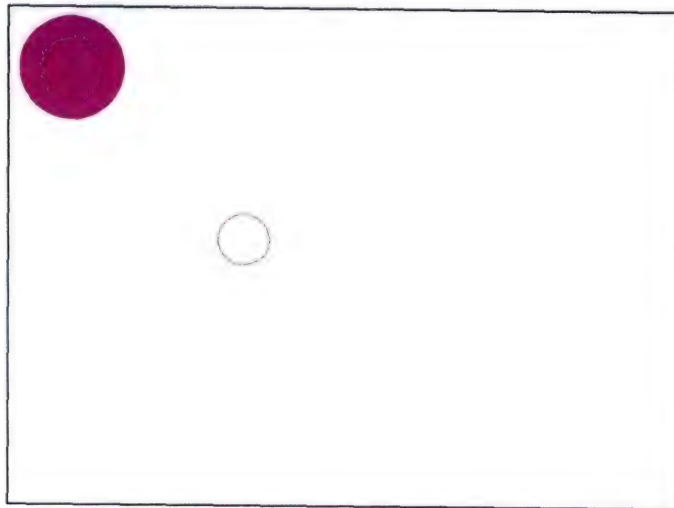


图16.12 椭圆方法示例

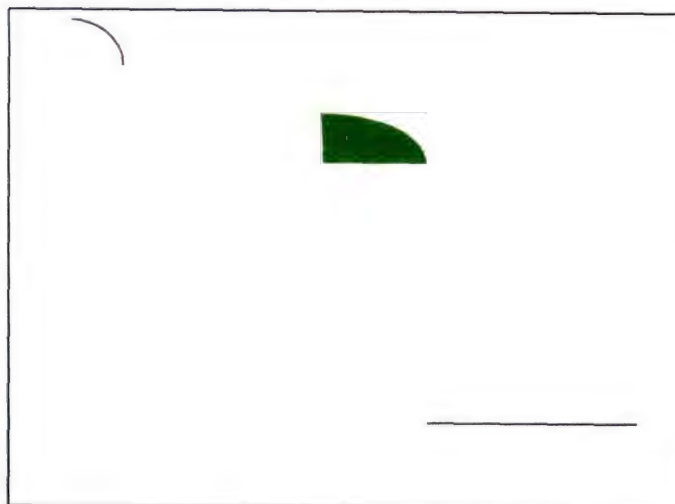


图16.13 圆弧方法示例

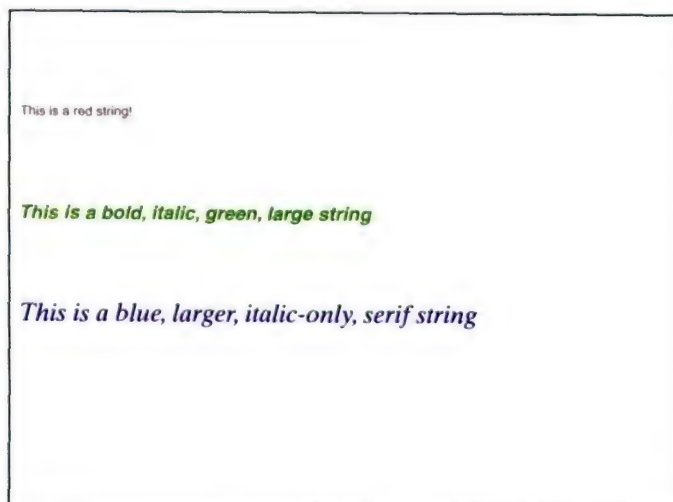


图16.14 文本方法示例

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域中取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭示了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短的现状下，美国等发达国家在其计算机科学发展的几十年间积淀和发展的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起到积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章公司较早意识到“出版要为教育服务”。自1998年开始，我们就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson、McGraw-Hill、Elsevier、MIT、John Wiley & Sons、Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum、Bjarne Stroustrup、Brain W. Kernighan、Dennis Ritchie、Jim Gray、Afred V. Aho、John E. Hopcroft、Jeffrey D. Ullman、Abraham Silberschatz、William Stallings、Donald E. Knuth、John L. Hennessy、Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍。其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方式如下：

华章网站：www.hzbook.com

电子邮件：hzjsj@hzbook.com

联系电话：(010) 88379604

联系地址：北京市西城区百万庄南街1号

邮政编码：100037



华章科技图书出版中心

译者序

Introduction to Computing and Programming in Python: A Multimedia Approach, 2E

这是一本什么书

这是一本什么样的书呢？根据主要内容，我们不妨先给出3种可能的诠释：

- 1) 一本讲Python编程的书，以多媒体处理为例增强趣味性；
- 2) 一本讲多媒体编程的书，选用Python作为示例语言；
- 3) 一本别出心裁的“计算机导论”教程，以Python多媒体编程为线索讲解各种有趣的计算机科学知识，把读者带进充满乐趣的计算机科学殿堂。

每个答案都不算错，但最恰当的当数第3个。没错，本书以Python数字多媒体编程为主线，依次讲解了图像、声音、文本和电影的处理，其中穿插介绍了大量的计算机科学基础知识。方法独到，示例通俗，条理清晰，将趣味性和实用性融于讲解之中，所有这些都会给阅读本书带来美好的体验。

除了主要内容（数字多媒体处理）之外，本书的最后一部分还介绍了更通用的计算机科学与编程知识：计算机性能、函数式编程和多媒体编程。

本书读者对象

显然，本书适合用做计算机专业导论课或者非计算机专业编程课程的教材。

想了解或温习一下计算机数字多媒体处理知识的专业程序员也可以翻翻这本书。

想学习如何用程序处理多媒体的非专业人士也可以读一读。新东方的李笑来老师因为“能够编写一些批处理脚本”，才有了《TOFEL核心词汇21天突破》。想学习编程的朋友们，可别忘了Python是非常强大、非常流行，同时又易懂易学的脚本语言。

最后，所有对Python语言感兴趣，想通过一些简单实用的例子来学习这门编程语言的朋友，都可以看看这本书。

我与华章

自从跟聂雪军、王昕等朋友合作翻译《代码之美》起，这是我与机械工业出版社华章公司的第5次合作了，非常感谢他们五六年来对我一贯的信任，特别是陈冀康先生。

华章一直在大量引进国外的优秀计算机图书，我本人也是受益者，最近正在精读他们引进的《Computer Systems: A Programmer's Perspective》第2版。

致谢

首先感谢我的家人在这4个多月的时间里给予的支持。

本书的审校工作再次邀请了校对奇人张伸（新浪微博：@loveisbug）帮忙，此人眼尖、心细，再加上因博览群书而培养起来的文字感，每次都能帮上大忙。上次审校《Java语言精粹》一书，出版后发现的唯一一处错别字出现在未经他过目的“译者序”中。这次他又帮忙通读了全书译稿（包括译者序），十分感谢他的辛勤和一丝不苟。

遇到较难翻译的内容时，译者经常上译言网（www.yeeyan.org）论坛求助。感谢耐心解答过问题的以下网友：nc、桥头堡、dingdingdang、qmiao和sparklark。

在我的gtalk上，高远（新浪微博：@狼大人）是随叫随到的翻译顾问。谢谢他的不厌其烦和及时响应。

齐艳昀也为本书的翻译出了一份力，同样表示感谢。

最后不得不提及的是：初译本书时，有超过70%的内容是利用上下班时间在地铁上靠笔译完成的。感谢上海地铁2号线和7号线全体员工，有了你们，才有了平稳运行的地铁，让我可以平稳地做翻译。

联系译者

尽管译者尽心尽力试图译好每一句话、每一个词，但限于时间和水平，错误疏漏在所难免，恳请读者朋友多多批评，不吝赐教。

勘误信息可以提交到译者个人的豆瓣小站上：<http://site.douban.com/120940>，也可以直接发往译者的邮箱：steedhorse@163.net，还可以关注译者的新浪微博：@steedhorse，直接互动。

王江平

2011年8月 于上海浦东

第2版前言

Introduction to Computing and Programming in Python: A Multimedia Approach, 2E

感谢大家对本书第1版的兴趣，正是大家的热情支持我们完成了第2版。感谢那些不知道名字的朋友提供的阅读评论，这些评论指导我们完成了本次修订。

第1版和第2版的主要区别如下：

- 1) 增加了对Python语言的覆盖面，包括更多标准库方面的话题、全局域和更多的控制结构。
 - 2) 更加强调了抽象代码和可复用代码。
 - 3) 第13章增加了一些内容，指导大家制作标准的AVI或QuickTime电影与他人分享。
 - 4) 每章结尾处大大增加了练习题的数量。
 - 5) 所有的下标都改为从0开始，而不是从1开始。使用0，而不是1作为第一个下标，与标准Python更兼容。
 - 6) 去掉了使用Swing创建用户界面和介绍JavaScript的一章。
 - 7) 重写了关于设计和调试的一章，加入了设计和测试示例，强调了维护。
 - 8) 把创建和修改文本的一章拆成了两章。增加了有关隐写术（steganography）的例子。
 - 9) 把关于语言范式（编程风格）的一章拆成了两章，以提供更多关于函数式编程（比如非可变函数：non-mutable function）和面向对象编程（使用与Logo中类似的小海龟来介绍对象）的内容。
 - 10) 增加了更多教师们希望在导论课中提及的概念，比如负数的二进制表示。
 - 11) 整体上，语言描述更加清晰，删掉了一些不必要的细节。（希望如此。）
 - 12) 所有软件和素材的最新版本可从<http://mediacomputation.org>网站找到。
- 第2版增加了一位合著者为本书的写作出版带来了美好的回忆。

Mark Guzdial和Barbara Ericson

佐治亚理工学院

计算机教育方面的研究表明：人们不只是“学习编程”，而是学习编程去实现某种事物[5, 22]，而做事动机的不同会决定人们是否学习编程[8]。每一位教师都面临着一个挑战：挑选某种事物来激发学习编程的兴趣，且要有足够强大的激发效果。

人们需要交流，交流的欲望是首要兴趣之一。人们逐渐将计算机更多地用做交流工具，而不是计算工具。今天，所有公开发行的文本、图像、声音、音乐和电影基本上都是用计算机技术制作的。

本书教大家如何编写程序，从而实现用数字媒体进行的交流。本书关注的是如何像专业程序员那样处理图像、声音、文本和电影，但使用的却是连学生都能写的程序。我们知道，大多数人会使用专业级的应用软件来实现这种类型的数字媒体处理。然而，懂得如何自己编写程序意味着你能做更多事情，而不是只能做手边的应用程序所支持的那些事情。你的表达能力不会受应用程序的限制。

还有一点：了解数字媒体应用程序中算法的工作原理会让你更好地使用它们，或者更方便地从一种应用程序转向另一种应用程序。对应用程序来说，如果你关心的是哪个菜单项做哪些事情，那么每种应用程序都是不同的。相反，如果你关心的是按自己喜欢的方式来调整像素的位置和颜色，那么，透过菜单项关注自己想要表达的东西可能会更容易。

本书不只讲数字媒体编程。处理数字媒体的程序实现起来并不容易，即使实现了其行为也可能出乎你的意料。问题自然就来了，比如“同样的图像为什么在Photoshop中做滤镜操作速度更快？”或者“这太难调试了——有更容易调试的编程方法吗？”回答这样的问题是计算机科学家的事情。本书末尾有好几章讲到了计算，而不只是编程。最后的这几章内容从数字媒体处理延伸到了更一般的话题。

计算机是人类设计出来的、最令人惊叹的创造性设备。它完全由精神素材（mind-stuff）构成。“与其梦想它，不如成为它”的理念在计算机上完全可行。只要能想象一件事，就可以让它在计算机上成为“现实”。玩转编程可以是、也应该是极大的乐趣。

致教师

本书的内容符合ACM/IEEE计算机课程2001（Computing Curriculum 2001）标准文档[4]中描述的“从命令式开始”的要求。内容从赋值、顺序操作、迭代、条件式和函数定义这样的基础内容开始。当学生掌握这些内容之后，才强调抽象内容（如算法复杂度、程序效率、计算机组成、层次式分解、递归和面向对象编程）。

这种非常规的教学次序依据的是学习科学（learning science）中的研究发现：记忆是关联性的。我们记忆新的东西靠的是把它跟旧的东西关联起来。基于“将来某天可能有用”的假定，人们能学会概念和技巧，但这些概念和技巧只跟那个假定相关联。结果就像所谓的“脆弱知识”（brittle knowledge）[9]——这种知识能让你通过考试，但你很快会忘记它，因为除了在那课堂上听过它之外，没有其他事情与之关联了。

如果概念和技巧能跟许多不同的思想关联起来,或者跟日常生活中的想法关联起来,那么记忆就会更牢固。想让学生获得可转移的知识 (transferable knowledge, 即可在新情形中运用的知识), 我们必须帮助他们把新知识和更一般的问题关联起来, 这样, 记忆就可以通过关联这些问题来索引它们[26]。本书中, 我们通过学生可以考察、关联 (比如在消除照片红眼效果时使用的条件式), 之后再在上面叠加抽象 (比如分别使用递归和函数式滤镜加映射来达到同样的目标) 的具体体验来讲解计算与编程。

我们知道, 对学习计算机的学生来说, 从抽象起步效果并不好。Ann Fleury揭示了这样的事实: 如果讲封装和复用[13], 那么计算机导论课上的学生根本听不懂。学生们更喜欢便于跟踪的简单代码, 而且真正认为这样的代码更好。学生们需要时间和经验来理解设计良好的系统所包含的价值。没有经验, 学生们很难学习抽象。

本书采用的多媒体计算教学方法从一些常见任务开始, 如图像处理、考察数字音乐、查看和制作网页、制作视频等, 许多人都用计算机做这些事。然后, 我们基于这些活动解释编程和计算。我们想让学生在访问Amazon (打个比方) 网站时会想到: “这是个网上书店——我知道它是用数据库和把数据库实体格式化成网页的一组程序来实现的。”我们想让学生在使用Adobe Photoshop和GIMP时, 思考图像滤镜是如何实际处理一个像素的红、绿、蓝颜色分量的。从一个有意义的上下文开始, 知识和技能更容易传递。它也使书中的例子更有意思、更能激发学生兴趣, 这些都有助于学生上课时集中精力。

基于多媒体计算的教学方法将大约2/3的时间花在多媒体的体验上, 在一个能调动学习兴趣的环境中让学生获得各种不同媒体的经验。在这2/3以后, 他们自然会问一些与计算有关的问题, 典型的问题如“为什么Photoshop比我的程序快?”或者“电影的代码很慢——为什么会这么慢?”此时, 我们会引入抽象和来自计算机科学的真知灼见来回答他们的问题。那是本书最后一部分的内容。

另外一个计算机教育领域的研究团体考察了计算机导论课的退课率和失败率居高不下的原因。一种常见的论点是: 计算机课程看上去“不切实际”, 而且过分关注诸如效率之类“单调乏味的细节”[1, 31]。然而, 学生们认为 (在问卷调查和面谈中告诉我们的) 与交流有关的上下文是切合实际的。这个切合实际的上下文部分地解释了我们在佐治亚理工学院 (Georgia Tech) 导论课程上的长期成功的原因, 本书就是为这一课程写就的。

在本书采用的教学方法中, 知识次序的不同不只体现在最后介绍抽象这一点上。我们在第3章的第一个重要程序中就开始使用数组和矩阵。通常, “计算机导论”课程都会把数组的知识推到后面, 因为它们明显比保存简单数值的变量复杂。一种既切合实际又具体可行的上下文是非常强大的[22]。我们发现学生们处理图片中的像素矩阵完全没有问题。

据统计, “计算机导论”课程中, 学生们退课或得到D、F分数的比率 (通常称为WDF率) 处在30%~50%的范围或更高。根据最近一项针对计算机导论课程失败率的全球性调查, 美国54所大学的平均失败率为33%, 17所国际大学的平均失败率为17%[6]。在佐治亚理工学院, 从2000年到2002年, 所有专业要求的导论课程中, 平均WDF率为28%。我们在自己的“媒体计算导论” (Introduction to Media Computation) 课程中使用了本教程的第1版。首次试开这门课的时候, 有121名学生选修, 其中没有计算机或工程专业的学生, 而且2/3是女生。最终我们的WDF率只有11.5%。

之后的两年里 (从2003年春到2005年秋), 我们的课程在佐治亚理工学院的平均WDF率 (多位讲师, 数千名学生) 为15%[21]。事实上, 之前28%的WDF率与现在15%的WDF率是没

有可比性的，因为所有专业都采用了之前的课程，只有人文、建筑和管理专业采用了新课程。个别专业取得了更显著的变化。比如，管理专业在1999~2003年采用较早的课程时，WDF率为51.5%，而采用新课程的两年中只有11.2%的失败率[21]。自本书第1版发行之后，其他几所学校也采用了这种方法并根据自身的情形做了调整而且评估了结果，他们在成功率上都取得了类似的显著进步[36, 37]。

怎样使用本书

本书以基本一致的次序呈现了我们在佐治亚理工学院的授课内容。个别老师可能会跳过某些章节（比如有关加法合成、MIDI和MP3的部分），不过，所有的内容我们都在自己的学生身上试讲过的。

然而，本书的使用并非仅限于此，还有人以其他方式使用过它：

- 只用第2章和第3章就可以对计算做个简单介绍，或者还需要第4章、第5章的一些内容。甚至有人用本书讲过只有一天的媒体计算专题研讨会。
- 第6~8章基本上重复了第3~5章中的计算机科学概念，但上下文是声音而不是图像。我们发现这种重复很有用——有些学生似乎在使用一种媒体时比使用另一种能更好地关联迭代和条件式的概念。此外，它让我们有机会指出同一种算法可在不同的媒体中拥有类似的效果（比如，缩放图片与调整音高用的是同一种算法）。但如果要节省时间，当然可以跳过这部分。
- 第12章在编程和计算方面没有介绍新的概念。电影的处理尽管能激发学生的兴趣，但为了节省时间也可以跳过。
- 关于最后一部分，我们建议有些章节至少要讲一讲，从而引导学生以更抽象的方式思考计算和编程，但显然没必要涵盖所有章节。

Python和Jython

本书使用的编程语言是Python。人们把Python描述为“可执行的伪代码”（executable pseudo-code）。我们发现计算机专业和非专业人士都可以学习Python。由于Python实际用于交互任务（比如网站开发），因此它很适合用做“计算机导论”课程的语言。例如，Python网站（<http://www.python.org>）上的招聘广告显示：像谷歌和Industrial Light & Magic这样的公司都在招Python程序员。

具体来讲，本书使用的Python“方言”是Jython（<http://www.jython.org>）。Jython就是Python。Python（通常用C实现）和Jython（用Java实现）的区别类似于任何两种语言实现之间的区别（比如Microsoft和GNU C++的实现）——基础语言是完全一致的，区别仅在于某些库和细节方面，大多数学生都注意不到。

格式说明

Python示例代码的字体如：`x = x + 1`。更长的示例版式如下：

```
def helloWorld():
    print "Hello, world!"
```

当显示用户输入对Python响应的内容时，字体风格类似，但用户的输入将出现在Python提示符（>>>）之后：

```
>>> print 3 + 4
```

```
7
```

在本书中，你还会发现几种特殊类型的图标。



计算机科学思想

关键的计算机科学概念以此种图标显示。



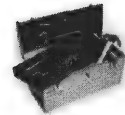
常见bug

可能导致程序失败的常见问题以此种图标显示。



调试技巧

事前预防bug潜入程序的好办法以此种图标突出显示。



实践技巧

真正有帮助的最佳实践或技术以此种图标突出显示。

致谢

衷心感谢：

- Jason Ertle、Claire Bailey、David Raines和Joshua Sklare，他们在非常短的时间内以惊人的高质量做出了第1版的JES。多年来，Adam Wilson、Larry Olson、Yu Cheung (Toby) Ho、Eric Mickley、Keith McDermott、Ellie Harmon、Timmy Douglas、Alex Rudnick、Brian O'Neill和William Fredrick (Buck) Scharfnorth III使JES成为今天这个简单实用且仍然易于理解的工具。
- Adam Wilson构建了对研究声音和图像以及处理视频来说非常有用的MediaTools。
- Andrea Forte、Mark Rickman、Matt Wallace、Alisa Bandlow、Derek Chambless、Larry Olson和David Rennie帮忙整理了教程讲义。Derek、Mark和Matt编写了许多示例程序。
- 许多佐治亚理工学院人共同努力成就了这项工作。副教务长Bob McMath和计算机学院的教务副主任Jim Foley很早就投入了这项工作。Kurt Eiselt努力使这项工作变成现实，说服他人认真对待这项工作。“为初涉计算机科学的学生讲授媒体计算”，Janet Kolodner和Aaron Bobick对这种想法感到兴奋和鼓舞。Jeff Pierce审查了书中所用媒体函数的设计并给出了建议。关于如何精确地传达数字化素材的内容，Aaron Lanterman提出了很多建议。Joan Morton、Chrissy Hendricks、David White，以及GVU中心的所有工作人员都确保满足我们的需要并处理好全部细节，确保工作向着同一个目标前进。Amy Bruckman和Eugene Guzdial为Mark争取了时间，使最终版本得以完成。
- 非常感谢Colin Potts和Monica Sweat，他们在佐治亚理工学院讲这门课，为我们提供了许多与课程有关的见解。

- Charles Fowler是佐治亚理工学院之外冒险在自己的学校（盖恩斯维尔学院）尝试这门课程的第一人，对此我们深表谢意。
- 2003年春季，佐治亚理工学院开设的试讲课程对这门课程的改进非常重要。Andrea Forte、Rachel Fithian和Lauren Rich评估了这次试讲，这让我们理解哪些地方有效果、哪些地方没有效果具有不可低估的价值。第一批助教（Jim Gruen、Angela Liang、Larry Olson、Matt Wallace、Adam Wilson和Jose Zagal）为这一教学方法的设计做了大量工作。Blair MacIntyre、Colin Potts和Monica Sweat帮忙调整了教程讲义，从而方便其他人使用。Jocken Rick把CoWeb/Swiki变成了CS1315课上学生们最爱逛的好地方。
- 许多学生指出了书中的错误并提出了改进建议。感谢Catherine Billiris、Jennifer Blake、Karin Bowman、Maryam Doroudi、Suzannah、Gill、Baillie Homire、Jonathan Laing、Mireille Murad、Michael Shaw、Summar Shoaib，特别是Jonathan Longhitano，他在文字编辑方面很有天赋。
- 感谢之前媒体计算课上的学生Constantino Kombosch、Joseph Clark和Shannon Joiner同意我在示例中使用他们的课堂快照。
- 与本书相关的研究工作得到了国家科学基金会（National Science Foundation）的赞助——赞助来自本科教育部门的CCLI项目，以及CISE教育革新项目。感谢你们的支持。
- 感谢计算机课上的学生Anthony Thomas、Celines Rivera和Carolina Gomez允许我们使用他们的图片。
- 最后，但最重要的，感谢我们的孩子Matthew、Katherine和Jenifer Guzdial，他们同意了爸爸妈妈的媒体项目而被拍照、录音，他们对这门课程非常支持并为之欢欣鼓舞。

Mark Guzdial和Barbara Ericson

佐治亚理工学院

目 录

Introduction to Computing and Programming in Python: A Multimedia Approach, 2E

出版者的话
译者序
第2版前言
第1版前言

第一部分 导 论

第1章 计算机科学与媒体计算导论	2
1.1 计算机科学是关于什么的	2
1.2 编程语言	4
1.3 计算机理解什么	5
1.4 媒体计算：为什么要把媒体数字化	7
1.5 大众的计算机科学	8
1.5.1 计算机科学与交流有关	8
1.5.2 计算机科学与过程有关	9
习题	9
第2章 编程导论	11
2.1 编程与命名有关	11
2.2 Python编程	13
2.3 JES编程	13
2.4 JES媒体计算	15
2.4.1 显示图片	18
2.4.2 播放声音	19
2.4.3 数值命名	20
2.5 构建程序	22
习题	27
第3章 使用循环修改图片	30
3.1 图片的编码	30
3.2 处理图片	35
3.3 改变颜色值	40
3.3.1 在图片上运用循环	40
3.3.2 增/减红（绿、蓝）	42
3.3.3 测试程序：它真的能运行吗	45
3.3.4 一次修改一种颜色	46
3.4 制作日落效果	47

3.5 亮化和暗化	51
3.6 制作底片	52
3.7 转换到灰度	53
习题	55
第4章 修改区域中的像素	58
4.1 复制像素	58
4.2 图片镜像	60
4.3 复制和转换图片	66
4.3.1 复制	66
4.3.2 制作拼贴图	72
4.3.3 通用复制	74
4.3.4 旋转	75
4.3.5 缩放	77
习题	81
第5章 高级图片技术	84
5.1 颜色替换：消除红眼、深褐色调和 色调分离	84
5.1.1 消除红眼	86
5.1.2 深褐色调和色调分离：使用条件式 选择颜色	88
5.2 合并像素：图片模糊化	92
5.3 比较像素：边缘检测	93
5.4 图片融合	94
5.5 背景消减	96
5.6 色键	98
5.7 在图像上绘图	101
5.7.1 使用绘图命令	102
5.7.2 向量和位图表示	104
5.8 指定绘图过程的程序	105
习题	107

第二部分 声 音

第6章 使用循环修改声音	110
6.1 声音是如何编码的	110

6.1.1 声音的物理学	110	第9章 构建更大的程序	164
6.1.2 探索声音的样子	113	9.1 自顶向下设计程序	164
6.1.3 声音编码	115	9.1.1 自顶向下设计示例	165
6.1.4 二进制数和二进制补码	116	9.1.2 设计顶层函数	166
6.1.5 存储数字化的声音	117	9.1.3 编写子函数	168
6.2 处理声音	118	9.2 自底向上设计程序	171
6.2.1 打开声音并处理样本数据	118	9.3 测试程序	172
6.2.2 使用JES媒体工具	121	9.4 调试技巧	174
6.2.3 循环	123	9.4.1 找出担心的语句	174
6.3 改变音量	123	9.4.2 查看变量	175
6.3.1 增大音量	123	9.4.3 调试冒险游戏	176
6.3.2 真的行吗	124	9.5 算法和设计	179
6.3.3 减小音量	127	9.6 在JES之外运行程序	180
6.3.4 理解声音函数	128	习题	181
6.4 声音规格化	128		
习题	131	第三部分 文本、文件、网络、	
第7章 修改一段样本区域	133	数据库和单媒体	
7.1 用不同方法处理不同声音片段	133	第10章 创建和修改文本	186
7.2 剪接声音	135	10.1 文本作为单媒体	186
7.3 通用剪辑和复制	140	10.2 字符串：构造和处理字符串	187
7.4 声音倒置	142	10.3 处理部分字符串	189
7.5 镜像	143	10.3.1 字符串方法：对象和点号语法	
习题	144	简介	190
第8章 通过合并片段制作声音	146	10.3.2 列表：强大的结构化文本	191
8.1 用加法组合声音	146	10.3.3 字符串没有字体	194
8.2 混合声音	147	10.4 文件：存放字符串和其他数据的	
8.3 制造回声	148	地方	194
8.3.1 制造多重回声	149	10.4.1 打开文件和操作文件	195
8.3.2 制作和弦	149	10.4.2 制作套用信函	197
8.4 采样键盘工作原理	150	10.4.3 编写程序	197
8.5 加法合成	153	10.5 Python标准库	201
8.5.1 制作正弦波	153	10.5.1 再谈导入和私有模块	202
8.5.2 把正弦波叠加起来	155	10.5.2 另一个有趣模块：Random	202
8.5.3 检查结果	156	10.5.3 Python标准库的例子	204
8.5.4 方波	157	习题	205
8.5.5 三角波	158	第11章 高级文本技术：Web和信息	208
8.6 现代音乐合成	160	11.1 网络：从Web获取文本	208
8.6.1 MP3	161	11.2 通过文本转换不同媒体	211
8.6.2 MIDI	161	11.3 在图片中隐藏信息	216
习题	162	习题	219

第12章 产生Web文本221

12.1 HTML：Web的表示方法221

12.2 编写程序产生HTML225

12.3 数据库：存放文本的地方229

12.3.1 关系型数据库231

12.3.2 基于散列表的关系型数据库示例...231

12.3.3 使用SQL.....234

12.3.4 使用数据库构建Web页面236

习题237

第四部分 电 影

第13章 制作和修改电影240

13.1 产生动画241

13.2 使用视频源247

13.3 自底向上制作视频效果250

习题254

第五部分 计算机科学议题

第14章 速度258

14.1 关注计算机科学258

14.2 什么使程序速度更快258

14.2.1 什么是计算机真正理解的258

14.2.2 编译器和解释器259

14.2.3 什么限制了计算机的速度263

14.2.4 让查找更快265

14.2.5 永不终止和无法编写出的算法 ...266

14.2.6 为什么Photoshop比JES更快.....268

14.3 什么使计算机速度更快268

14.3.1 时钟频率和实际的计算268

14.3.2 存储：什么使计算机速度慢269

14.3.3 显示270

习题270

第15章 函数式编程272

15.1 使用函数简化编程272

15.2 使用Map和Reduce进行函数式编程 ...275

15.3 针对媒体的函数式编程277

15.4 递归：一种强大的思想279

15.4.1 递归式目录遍历284

15.4.2 递归式媒体函数286

习题287

第16章 面向对象编程289

16.1 对象的历史289

16.2 使用“小海龟”290

16.2.1 类和对象290

16.2.2 创建对象290

16.2.3 向对象发送消息291

16.2.4 对象控制自己的状态292

16.2.5 小海龟的其他函数293

16.3 教小海龟新的技艺295

16.4 面向对象的幻灯片297

16.4.1 Joe the Box.....300

16.4.2 面向对象的媒体302

16.4.3 为什么使用对象306

习题307

附录A Python快速参考309

参考文献313

导 论

第1章 计算机科学与媒体计算导论

第2章 编程导论

第3章 使用循环修改图片

第4章 修改区域中的像素

第5章 高级图片技术

计算机科学与媒体计算导论


本章学习目标

- 计算机科学是关于什么的和计算机科学家关心什么的。
- 我们为什么要把媒体数字化。
- 为什么学习计算 (computing) 是有价值的。
- 编码 (encode) 的概念。
- 计算机的基础组件。

1.1 计算机科学是关于什么的

计算机科学是关于过程 (process) 的学习：我们，或者计算机如何做事情，我们如何指定要做的事情，如何指定要处理的东西。这是个相当枯燥的定义。让我们试着给出一个比喻性的描述吧。

计算机科学思想：学习计算机科学就像研究菜谱 (recipe)



它们是一类特殊的菜谱——一种可被计算设备执行的菜谱，但这一点只对计算机科学家有意义。总体而言，重要的一点是：计算机科学的菜谱精确地定义了必须完成的工作。

如果你是一位生物学家，想要描述生物迁徙或DNA复制的原理，那么以一种可被全面定义和理解的方式编写一套菜谱，精确地说明所发生的事情，将很有帮助。如果你是一位化学家，想要解释化学反应中如何达到平衡，情况也一样。使用计算机程序，工厂经理可以定义机器和皮带的布局，甚至测试它如何工作——而不必真正把那些重家伙搬到实际位置上。可以在计算机上运行的菜谱称为程序。计算机之所以能对科学的研究和理解带来如此深刻的变化，精确定义任务和模拟事件的能力是其首要原因。

用菜谱来比喻程序听起来有点儿滑稽，但这个比喻非常有用。计算机科学家研究的很多东西都可以用菜谱来形容：

- 有些计算机科学家研究如何编写菜谱：完成一项任务有更好或更差的方法吗？如果你曾经试图把蛋清跟蛋黄分开，那么就会明白是否懂得正确方法结果大不相同。计算机科学的理论家关注最快和最短的菜谱，以及占用空间最少的菜谱（可以把它想象成空间对抗——这是个不错的比喻）。菜谱如何工作与如何编写菜谱完全不同，这方面的工作称为算法研究。软件工程师关注大的团队如何把菜谱合并起来且仍然正常工作。（某些程序，就像跟踪Visa/MasterCard记录的那种，其菜谱包含上百万的步骤！）术语软件指的是完成一项任务的计算机程序的集合。
- 有些计算机科学家研究菜谱中使用的单位。菜谱使用公制度量单位还是英制度量单位重要吗？使用任何一种都可以，但如果不知道一磅或一杯是多少，那么对你来说菜谱就不

那么好理解了。也有些单位对某种任务有意义而对其他任务没有意义，但如果能将任务与使用的单位匹配好，那么你就能更方便地解释自己的想法，更快速地完成——并且避免错误。有没有想过大海中的轮船为什么使用海里/小时“knot”来度量速度？为什么不用类似“米/秒”这样的单位？有时在某种特殊情况下（比如在大海中的轮船上）常见的术语并不适合或者根本不好用。计算机科学中对单位的研究称为**数据结构**。有些科学家研究如何跟踪基于大量不同单位的大量数据，他们研究的是**数据库**。

- 任何事情都可以写成菜谱吗？是否有些菜谱是写不出来的？计算机科学家知道有些菜谱是写不出来的。比如，你无法写出一份菜谱来精确判断另外一些菜谱是否有实际效果。智能又是怎样的情况呢？我们能写出一份菜谱，让遵循它的计算机做到真正思考吗（你又如何判断它是否正确呢）？**计算机原理、智能系统、人工智能和计算机系统**方面的科学家关注的就是这类东西。
- 甚至有些计算机科学家关注人们是否喜欢菜谱给出的结果，就像报社的餐饮评论家。其中有一些是**人机界面**（产生人们使用的某种界面的“菜谱”，这类界面包括窗口、按钮、滚动条，以及我们能想到的运行中的程序所使用的其他元素）专家，关心人们是否喜欢菜谱工作的方式。
- 正如某些厨师专长于某一类菜谱，如烤饼或烤肉，有些计算机科学家也会专攻特定种类的菜谱。致力于图形领域的科学家主要关注产生图片、动画甚至电影的菜谱。致力于计算机音乐领域的科学家主要关注产生声音的菜谱（常常是有旋律的声音，但也不一定）。
- 还有一些计算机科学家研究菜谱的**突现特性（emergent property）**。考虑一下万维网。万维网实际上是由数百万相互关联的菜谱（程序）组成的集合。为什么网络的一部分在某一点上会变慢？这是数百万程序中出现的现象，当然不是人们预期的。这些是**网络计算机科学家**研究的内容。真正令人惊讶的是：这些突现特性（仅当你有许多同时交互的菜谱时才会发生的事情）也可以解释非计算机领域的事情。举例来说，蚂蚁寻食、白蚁筑堆，这些都可以描述成一种特殊的事情：它们仅发生在有大量的小程序在做简单、交互式的任务时。

换一个角度来考虑，菜谱的比喻仍然恰当。大家都知道菜谱中有些东西可以改变但不会使结果产生太大变化。你可以将所有单位增大一个因子（比如加倍）来产出更多的东西。在意大利面酱中加入更多的大蒜或牛至粉（oregano）。但菜谱中也有一些东西是不能改变的。如果菜谱中需要发酵粉，那么就不能以小苏打代之。如果打算把饺子煮熟再煎一下，顺序反过来恐怕也不会有好结果（如图1.1所示）。

同样的道理也适用于软件菜谱。通常，有些东西很容易改变：事物的实际名字（尽管你应该按一致的方式改变名字）、某些**常量**（以普通数字形式，而非变量形式出现的数值），或者还有待处理数据的某些**范围**（数据片段）。然而，发给计算机的命令，通常必须严格保持给定的次序。随着我们的讲解，你将学会哪些东西可以安全地改变，哪些不可以。

砂锅鸡	
剔骨鸡胸肉3整块	切碎的番茄1听（28盎司）
中等大小的洋葱1个，切碎	番茄汁1听（15盎司）
大蒜碎末1汤匙	蘑菇1听（6.5盎司）
橄榄油2汤匙，之后1/4杯	番茄酱1听（6盎司）
面粉1.5杯	意大利面酱*1/2罐（26盎司）
Lawry调味盐1/4杯	意大利调味料3汤匙
青椒1个，切碎（可选），颜色随意	大蒜粉1茶匙（可选）
鸡肉切成1英寸见方的小块。翻炒洋葱和大蒜直至洋葱呈透明状。加入面粉和Lawry调味盐。按1：4～1：5的比例混合调味盐和面粉，多到足以涂满鸡肉表面。把切好的鸡肉和调味的面粉放进一个袋子，抖动一下使面粉涂在鸡肉表面上。把涂好的鸡肉放进洋葱和大蒜中。快速翻炒直至鸡肉呈褐色。放一点油，防止它们黏在一起或者炒焦；我有时会加1/4杯橄榄油。放入番茄、面酱、蘑菇和番茄酱（以及可选的青椒），炒熟。放入意大利调味料。我喜欢大蒜，因此通常也会放点大蒜粉，搅匀。因为加了面粉，所以各种酱会变得很黏。我通常用意大利面酱把它们打开，最多半瓶。慢炖20~30分钟。	

图1.1 一道菜谱——你可以放入双倍调味料，但多放一杯面粉鸡肉就分不开了，而且，不要尝试在放入番茄汁后再爆炒鸡肉

1.2 编程语言

计算机科学家使用编程语言编写菜谱（如图1.2所示）。针对不同目的，可以使用不同的编程语言。有些语言广泛流行，如Java和C++。有些语言冷门一些，如Squeak和T。另外一些是为了让计算机科学思想更易于理解，如Scheme或Python，但易于学习的事实并不总能使它们非常流行或者成为专家们构建更大更复杂菜谱时的首选。讲授计算机科学时，选择一门既易于学习又足够流行且有用，而学生有动力去学习的语言，权衡起来颇有难度。

```
Python/Jython
def hello():
    print "Hello World"

Java
class HelloWorld {
    static public void main( String args[] ) {
        System.out.println( "Hello World!" );
    }
}

C++
#include <iostream.h>

main() {
    cout << "Hello World!" << endl;
    return 0;
}

Scheme
(define helloworld
  (lambda ()
    (display "Hello World")
    (newline)))
```

图1.2 编程语言比较：一种常见的简单编程任务就是在屏幕上打印“Hello World!”这句话

计算机科学家为什么不使用像英语或者西班牙语这样的人类自然语言呢？问题在于自然语言是沿着增进非常聪明的物种（人类）之间的沟通这条路发展起来的。正如我们将在下一节详细解释的那样，计算机是十分愚蠢的。它们对明确程度的要求，自然语言并不擅长。而且，我们在自然交流中彼此说的话与你在一套计算菜谱中说的话并不完全相同。你什么时候与人讲述过像毁灭战士（Doom）、雷神之槌（Quake）或超级玛丽兄弟（Super Mario Brothers）这种视频游戏的详细情节，详细到对方能复制出这款游戏（比如在纸上）的程度？英语并不擅长这类任务。

因为需要编写的菜谱种类很多，所以编程语言的种类也很多。通常，用C语言写的程序快速高效，但也常常难读难写，而且需要一些与计算机关系更为密切的单元，而不是与鸟类迁徙、DNA或其他任何你想要编写的菜谱有关的。Lisp语言（以及像Scheme、T和Common Lisp这样的相关语言）非常灵活，很适合用来探索如何编写以前从没写过的菜谱，但Lisp与C这样的语言相比样子太奇怪了，以至于很多人都不用它，结果知道它的人自然就更少了。假如你想雇一百个人来做项目，与不那么流行的语言相比，找到一百个了解一种流行语言的人要容易得多——但这并不意味着对你的任务而言那门流行语言是最好的。

在本书中使用的编程语言是Python（更多信息参阅<http://www.python.org>）。Python是一门相当流行的编程语言，常用于Web或媒体编程。Web搜索引擎谷歌就使用了Python。媒体公司Industrial Light & Magic也使用Python。你可以从<http://wiki.python.org/moin/Organizations-UsingPython>获得一份使用Python的公司列表。Python易学易读，非常灵活，但效率一般。同样的算法分别用C和Python编写，C代码很可能更快。

本书使用的是称为Jython（<http://www.jython.org>）的Python版本。Python通常用C语言实现。Jython是用Java实现的Python——这意味着Jython实际是Java编写的程序。Jython支持跨多种计算机平台运行的多媒体程序。Jython是真正的编程语言，可用于重要的工作任务。你可以从Jython网站为自己的计算机下载某个Jython版本，然后就可以用它做各种事情。在本书里，我们将通过一种称为JES（Jython Environment for Students）的特殊编程环境来使用Jython，这种编程环境能使Jython编程更加方便。但JES中能做的任何事情，在一般的Jython中也都能做——而且，用Jython编写的大部分程序在Python中也能正常运行。

以下解释了在本书中使用的重要术语：

- **程序**是使用编程语言描述的过程，这种过程能达到对某人有用的某种结果。程序可以很小（就像计算器的程序），也可以很大（就像银行用来跟踪所有账户的程序）。
- **算法**（与程序不同）是独立于编程语言的过程描述。同样的算法可以在多种不同的程序中以多种不同的方式使用多种不同的语言实现，但同一种算法具有相同的过程。

本书中使用的术语**菜谱**描述完成某种任务的程序或程序的一部分。将使用术语**菜谱**来强调完成某种有用的与**媒体**有关的任务的程序片段。

1.3 计算机理解什么

编写计算菜谱是为了在计算机上运行。计算机如何知道要执行什么呢？使用菜谱，我们又能让计算机做什么呢？答案是：“能做的少之又少。”计算机格外愚蠢。实际上，它们只知道数字。

其实，说计算机知道数字都不完全准确。计算机使用数字的编码（encoding）。计算机是

响应线路电压的电子设备。每条线路称为一个位 (bit)。如果一条线路上有电压，那么我们就说它编码了一个“1”；如果没有电压，我们就说它编码了“0”。我们把这些线路 (位) 分组。一组8位称为一个字节 (Byte)。于是，通过一组8条线路 (一个字节)，我们就有了一种8个0或1的模式，比如，01001010。使用二进制 (binary) 数字系统，我们可以把这个字节解释为一个数字 (如图1.3所示)。我们说计算机知道数字，就是这么来的。

计算机有一段塞满字节的内存 (memory)。在任一时刻，计算机处理的东西都存储在它的内存中。这就是说计算机处理的任何东西都编码在字节之中——JPEG图片、Excel表格、Word文档、讨厌的Web弹出广告，还有上一封垃圾邮件。

计算机可以用数字做许多事情。它可以把数字加、减、乘、除、排序、收集、复制、过滤 (比如，“把这些数字复制一份，但只要偶数”。)，或者比较它们并基于比较的结果做事。举例来说，一份菜谱可以告诉计算机：“比较这两个数。如果第一个小于第二个，就跳转到本菜谱的第5步；否则，继续执行下一步。”

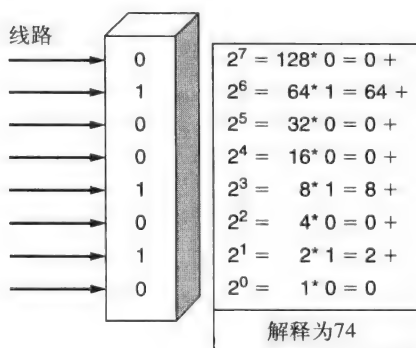


图1.3 具有电压模式的8条线路是一个字节，它被解释为8个0或1的一种模式，进而被解释为一个十进制数

说到现在，似乎计算机就是一种奇特的计算器，这当然是它被发明出来的原因。计算机最初的用途之一就是在第二次世界大战中计算抛射体弹道 (如果风来自东南方，风速每小时15英里，想击中北偏东30度0.5英里处的目标，那么弹筒应该倾斜到……)。现代计算机每秒钟能做几十亿次运算。然而，使计算机可用于通用菜谱的原因不在其他，恰在于编码的概念。

计算机科学思想：计算机可以把编码分层

计算机可以把编码分层，分到几乎任意复杂的程度。数字可以解释为字符，字符组合起来又可以解释成网页，而网页被解释之后，可以显现为多种字体和风格。但在最底层，计算机只“知道”被我们解释为数字的电压。

如果把某个字节解释为数字65，那么它可能就是数字65。或者，使用一种数字到字母的编码标准：美国标准信息交换编码 (American Standard Code for Information Interchange, ASCII)，它可以是字母A。如果65出现在被我们解释为文本的其他一组数字中，而且保存在以“.html”结尾的文件里，那么它可能是类似“<a href=...”结构的一部分，在Web浏览器中被解释为链接的定义。在计算机下层，那个A只是一种电压模式。菜谱一层层堆上去，在Web浏览器的层上，它就定义了你点击之以获取更多信息的某种东西。


如果计算机只理解数字 (这已经是一种延伸)，它如何处理这些编码呢？当然，它知道如

何比较数字，但它又怎样扩展这种能力从而按字母顺序排列一个类的列表呢？通常，每一层编码都实现为软件的一部分或一层。我们有懂得如何操控字符的软件。字符软件知道如何进行名字比较，因为它已经编码了诸如a在b之前这样的信息，而比较字母编码中的数字顺序，也就得到了字母顺序。字符软件又被其他软件使用，用来处理文件中的文本。像Microsoft Word、Notepad或TextEdit这样的软件就会用到这一层。另一种软件片段知道如何解释HTML（Web的语言），而同一软件的另外一层知道如何将HTML显示成正确的文本、字体、风格和颜色。

按照类似的方法，我们可以在计算机中为自己的任务创建编码层次。我们可以告诉计算机细胞包含线粒体和DNA，DNA有4种核苷酸，工厂里有这些种类的印刷机和那些种类的压印器等。创建编码和解释的层次，从而让计算机针对特定问题使用正确的单元，这是**数据表示**的任务，或者说定义正确的**数据结构**。

听上去似乎有很多软件，的确如此。软件按这种方式分层后，会使计算机的速度有所下降，但计算机的强大之处就在于它的速度奇快——而且一直在变得更快。

计算机科学思想：摩尔定律（Moore's Law）



Intel（Intel制造运行Windows操作系统的计算机上所使用的处理器）的创立者之一戈登·摩尔（Gordon Moore）提出：晶体管（计算机的关键组件）的数目每18个月会增加一倍，而价格保持不变；这意味着同样的钱每隔18个月就能买到双倍的计算能力。也就是说，计算机一直在变得更小、更快、更便宜。几十年了，这一定律一直成立。

如今的计算机每秒钟可以执行数十亿个菜谱步骤。它们能把百科全书的数据保存在内存中！它们不知疲倦，永不厌烦。在100万用户中查找某个持卡人？没有问题！找出使方程取最优值的数值集合？小菜一碟！

处理数百万图片元素、声音片段或电影画面？这就是**媒体计算**了。通过这本书，你将编写出处理图像、声音和文本的菜谱，甚至其他菜谱。这是可能的，因为在计算机中任何东西都是以数字化的形式表示的，菜谱也是。看完这本书，你将编写出实现数字视频特效的菜谱，用来创建Web页面，就像Amazon和eBay那样；以及像Photoshop那样对图像进行滤镜处理的菜谱。

1.4 媒体计算：为什么要把媒体数字化

让我们考虑一种适合图片的编码。把图片想象成由一个个的小点组成。这不难想象：使劲靠近显示器或电视屏幕，你就能看到眼前的图片本来就是由小点组成的。每个小点都有自己的颜色。物理学告诉我们：颜色可以描述为红、绿、蓝的总和。红色和绿色相加得到的是黄色。这三种颜色加在一起得到的是白色。三种颜色都关掉，得到的就是一个黑点。

如果把图片中的每个点编码为三个字节的集合，每个字节表示该点显示在屏幕上时红色、绿色和蓝色的数量，情况会怎样呢？收集许多这样的三字节集合来决定给定图片上的所有点，又会怎样呢？这是一种相当合理的表示图片的方式，基本上也是第3章中我们将使用的方式。

操作这些点（每个点称为一个**像素**或**图片元素**）可能需要大量的处理。在一幅图片中，你可能需要从计算机或网页上处理上千甚至数百万个点。但计算机并不厌烦，而且速度极快。

我们将使用的声音编码每秒钟的声音包含44 100个双字节集合（这两个字节称做一个样本）。一首三分钟的歌曲需要158 760 000个字节（立体声要再加一倍）。在这些样本上做任何操作都需要大量的处理。但以每秒钟10亿次操作的速度，你可以在很短的时间里对这些字节做很多操作。

建立这种编码需要把媒体改变一下。看一看现实世界：它并不是由你能看清的大量小点组成的。听一听声音：你能听出每秒钟有几千个声音小段吗？你听不出每秒钟内的声音小段，正是这一事实使编码的创建成为可能。我们的眼睛和耳朵是有局限的：只能感知这么多，只能感知这么小的东西。如果把一幅图像拆分成足够小的点，那么你的眼睛无法分辨出它其实不是连续的颜色流。如果把一段声音拆分成足够小的片段，你的耳朵也无法分辨出它其实不是连续的声音能量流。

将媒体编码成小片段的过程称为**数字化** (digitization)，有时也称为“going digital”。(根据《American Heritage Dictionary》) Digital的意思是：“数字的、与数字有关的或类似数字的，特别是手指。”(Of, relating to, or resembling a digit, especially a finger.) 将事物数字化就是把一种连续的、不可数的东西转化成可数的东西，好比用手指头。

对能力有限的人类感官来说，数字媒体如果做得好，感觉上与原来的媒体是一样的。唱片录音机（见过没？）连续用**模拟**信号将声音捕捉下来，照片也把光线作为持续流捕捉下来。有些人说他们能听出唱片跟CD的不同，但对我们的耳朵以及大多数测量工具来说，CD（数字化的声音）听起来是一样的，甚至更清晰。数码相机能以足够高的清晰度生成相片质量的图片。

为什么要把媒体数字化？因为那样一来媒体就更容易处理、精确复制、压缩和传输。举例来说，相片中的图像很难处理，但同样的图像经数字化之后处理起来就容易多了。本书就是关于如何利用并处理不断数字化的媒体世界——并在此过程中学习电脑计算的。

摩尔定律使得媒体计算可以作为引导性的入门主题。媒体计算依赖于计算机在大量字节上执行大量操作。现代计算机很容易做到这一点。即使用缓慢（但容易理解）的语言、低效（但易写易读）的菜谱，我们仍可通过媒体处理来学习计算。

处理媒体时应尊重作者的数字版权。在不违反公平使用法的前提下，为教学目的修改图像和声音是允许的。但是，分享或发行处理过的图像或声音就有可能侵犯所有者的版权了。

1.5 大众的计算机科学

为什么要通过编写媒体处理程序来学习计算机科学呢？为什么不想成为计算机科学家的人也应该学习计算机科学呢？对于通过媒体处理来学习计算的过程，你又如何会感兴趣呢？

如今，多数专业都会用到媒体处理：报纸、视频、磁带录音、摄影、绘画。慢慢地，这种处理过程都改用计算机来完成了。如今，媒体大都以数字化的形式存在。

我们使用软件来处理这些媒体。我们使用Adobe Photoshop处理图像，使用Audacity处理声音，还可能用Microsoft PowerPoint把我们的媒体组装成幻灯片。我们使用Microsoft Word处理文本，使用Netscape Navigator或者Microsoft Internet Explorer来浏览因特网上的媒体。

那么，为什么不想成为计算机科学家的人也应该学习计算机科学呢？为什么应该学习编程呢？学会使用所有这些优秀的软件还不够吗？后面的几节给出了这些问题的答案。

1.5.1 计算机科学与交流有关

数字媒体用软件来处理。如果只能使用别人制作的软件来处理媒体，那么你的交流能力就会受到限制。如果你想表达一件事情，而Adobe、Microsoft、Apple或其他公司的软件都不能帮你表达，那该怎么办呢？或者，想以一种它们所不支持的方式来表达，又该怎么办呢？如果懂得如何编写程序，即使亲自动手将花费更多时间，那么你也拥有了按自己喜爱的方式处理媒体的自由。

从一开始就学习这些工具，又会有怎样的效果呢？在与计算机打交道的全部岁月里，我们见证了太多种类的软件，来了又去了，诸如绘图软件、绘画软件、字处理软件、视频编辑软件等。你不能只学习一种工具，然后指望在整个职业生涯中使用它。如果了解这些工具的原理，那么你便拥有了核心的理解力，就可以从一种工具转向另一种工具。你能够基于算法，而不是工具来思考自己的媒体作品。

最后，如果制作媒体是为了Web、市场、印刷、广播或其他类似目的，那就有必要了解一些对媒体能做什么不能做什么的常识。对媒体消费者来说，知道媒体可以怎样处理，知道什么是真实的，什么只是一种把戏，就更重要了。如果了解媒体计算的基本知识，那么你对媒体便有了超越任何工具所能提供的更深层次的理解。

1.5.2 计算机科学与过程有关

1961年，Alan Perlis在麻省理工学院做的一次演讲中提出：计算机科学，更明确地说是编程，应该是通识教育（liberal education）的一部分[17]。Perlis是计算机科学领域的重要人物。计算机科学的最高奖项是ACM图灵奖，Perlis是该奖的首位获得者。他是软件工程领域的著名人物，美国大学的第一批计算机科学系当中有好几个是他创立的。

Perlis的观点可以用微积分来类比一下。一般认为，微积分是通识教育的一部分：并非所有人都学微积分，但如果你想接受良好的教育，通常至少应修满一学期的微积分。微积分研究的是比率（rate），在许多领域都很重要。正如本章之前所讲的，计算机科学研究的是过程。过程对几乎所有的领域都重要，从商业到科学，从医药到法律。从规范的角度理解过程对每个人来说都很重要。计算机实现的过程自动化改变了每一个行业。

最近，Jeannette Wing提出每个人都应该学习计算思维（computational thinking）[34]。她认为计算学科中讲授的技巧类型对所有学生来说都是关键的技巧。这正是Alan Perlis预言的：自动化计算将改变我们了解世界的方式。

习题

- 1.1 如今每一种职业都使用计算机。试着用Web浏览器和搜索引擎（如谷歌）找到与你的研究领域相关的计算机科学或计算站点。比如搜索“生物计算机科学”或“管理计算”。
- 1.2 上网查找一份ASCII码表：一张罗列了各个字符及其相应数字表示的表格。写出组成你名字的ASCII字符所对应的数字序列。
- 1.3 上网查找一份Unicode表。看看ASCII和Unicode之间的区别是什么？
- 1.4 考虑1.4节描述的图片表示：图片中每个点（像素）用三个字节分别表示该点颜色的红、绿、蓝分量。基于这种方法，表示一张 640×480 像素的图片（网上常见的图片尺寸）需要多少字节？表示一张 1024×768 像素的图片（常见的屏幕尺寸）又需要多少字节？（现在，你觉得所谓“300万像素”的数码相机是个什么概念？）
- 1.5 1位可以表示0或1。2位有4种可能的组合：00、01、10、11。4位或8位（1字节）有多少种不同的组合？2字节（16位）能表示多少数字？4字节又能表示多少数字？
- *1.6 如何基于字节表示一个浮点数？上网搜索一下“浮点”（floating point），看看能找到什么信息。
- 1.7 上网查一下Alan Key和《Dynabook》。Alan与媒体计算有什么关系？
- 1.8 上网查一下Grace Hopper。她对编程语言有怎样的贡献？

- 1.9 上网查一下Philip Emeagwali。他得过哪种计算机科学方面的奖项？
- 1.10 上网查一下Alan Turing。了解一下计算机能做什么和编码如何工作这些观念与他有什么联系。
- 1.11 上网查一下Harvard Computers。他们对天文学做过何种贡献？
- 1.12 上网查一下Adele Goldberg。她对编程语言有何种贡献？
- 1.13 上网查一下Kurt Gödel。关于编码，他做了哪些令人惊叹的事情？
- 1.14 上网查一下Ada Lovelace。在第一台机械式计算机建造出来之前，她做了哪些令人惊叹的事情？
- 1.15 上网查一下Claude Shannon。他为自己的硕士论文做了哪些工作？
- 1.16 上网查一下Richard Tapia。他为增进计算的多样性做过什么？
- 1.17 上网查一下Frances Allen。她得过哪种计算机科学方面的奖项？
- 1.18 上网查一下Mary Lou Jepsen。她在研究什么新技术？
- 1.19 上网查一下Ashley Qualls。她创建了什么价值百万美元的东西？
- 1.20 上网查一下Marissa Mayer。她是做什么的？

深入学习

James Gleick的《Chaos》（混沌）一书对突现特性做了更多描述——阐释了微小变化如何导致了显著结果，以及难以预见的交互为何能给设计带来意料之外的影响。

Mitchel Resnick的《Turtles, Termites and Traffic Jams: Exploration in Massively Parallel Microworlds》（海龟、白蚁和交通阻塞：大规模并行微观世界探索）一书[33]讲述了通过同时运行成百上千的微小过程（程序），并让它们彼此交互，就可以相当精确地描述蚂蚁、白蚁、甚至交通阻塞和黏液菌的行为。

《Exploring the Digital Domain》（探索数字领域）[3]是一本极好的计算科学入门书，其中有许多关于数字媒体的有用信息。

编程导论

本章学习目标

本章媒体学习目标:


- 制作并显示图片。
- 制作并播放声音。

本章计算机科学学习目标:

- 使用JES输入并执行程序。
- 创建并使用变量来保存值和对象, 比如图片和声音。
- 创建函数。
- 识别诸如整数、浮点数和媒体对象等不同类型(编码)的数据。
- 顺序执行函数中的操作。

2.1 编程与命名有关

计算机科学思想: 大部分编程工作都与命名有关




计算机可以将名字, 或者符号, 关联到几乎任何东西: 特定的字节; 一组字节组成的数值变量或一串字符; 像文件、声音或图片这样的媒体元素; 甚至更抽象的概念, 比如命名的菜谱(程序)或命名的编码方式(类型)。就像哲学家和数学家一样, 计算机科学家也认为某些名字的选取具有更高的品质: 命名方案(名字及其命名的事物)应当优雅、节省且好用。命名是一种抽象形式。我们使用名字来指代被命名的东西。

显然, 计算机本身并不关心名字。名字是为人服务的。如果计算机只是个计算器, 那么记住词语及其关联值之间的关系不过是浪费内存。但对人来说, 命名却是非常强大的机制。它允许你以自然的方式使用计算机, 这种方式甚至全面扩展了我们理解菜谱(过程)的方式。

编程语言实际上就是一组名字的集合, 计算机拥有这些名字的编码, 于是它们能让计算机完成我们期望的工作, 或者按我们期望的方式解释数据。在有些语言中, 基于它所使用的名字我们可以定义新的名字, 从而构造自己的编码层次。给变量赋值就是为计算机定义名字的一种方法。定义函数则是为菜谱命名。

程序由一组名字和它们的值构成, 其中有些名字拥有计算机指令类型的值(“代码”)。我们的指令将使用Python语言来指定, 结合以上两个定义可以得出: Python编程语言为我们提供了一组有用的名字, 这些名字对计算机是有意义的。而我们的程序就是从这些有用的名字中选出一些, 然后加上我们自己定义的名字, 把它们联合起来就可以告诉计算机我们想让它做什么。

计算机科学思想: 程序以人为本, 而不是以机器为本



记住: 名字只对人有意义, 而非对计算机有意义。计算机只接受指令。好的程序是对人有意义(可理解且好用)的程序。

名字有好也有坏。一组良好的编码和名字能让我们以自然的方式定义菜谱而无须解释太多。各种语言可以分别理解为一套名字和编码的集合。对某种任务来说,有些语言更适合。描述同样的菜谱,有的语言需要你编写更多代码——但有时这种“更多”会带来(对人来说)可读性更好的菜谱,有助于别人理解你要表达的东西。

哲学家和数学家所追寻的品质感非常相似,他们尝试用几个词来描述世界,追寻既能涵盖更多情形又能保证同道中人可以理解的优雅词句组合。这恰恰是计算机科学家也在尝试的事情。

对于菜谱中的单位和值(数据),其解释方式常常也需要命名。还记得我们在1.3节说过的吗?任何东西都被编码成字节,但字节可以解释为数字。在有些编程语言中,你可以显式指定某个值是byte,之后又让语言把它作为数字看待,即一个integer(有时也叫int)。类似地,你可以告诉计算机这些字节是一组数字的集合(整数数组)、一组字符的集合(字符串),甚至是更为复杂的浮点数(float)(任何具有小数点的数字)的编码。

在Python中,我们将显式告诉计算机如何解释我们的值,但我们不会告诉它某个名字只与某种编码关联。像Java和C++那样的语言是强类型(strongly typed)的,在那些语言中名字与特定的类型或编码牢固地关联起来。它们要求你指明这个名字只与整数关联,那个名字只与浮点数关联。Python仍然拥有类型(可通过名字来引用的编码方式),但不像Java和C++那么明确。Python也有保留字(reserved words),保留字是一些词语,你不能用它们给自己的东西命名,因为它们在语言中已有特定含义。

文件和文件名

编程语言不是计算机关联名字和值的唯一场所。计算机的操作系统负责管理磁盘上的文件,并把它们与名字关联起来。你熟悉或使用的操作系统可能包括Windows 95、Windows 98(Windows ME、NT、XP、Vista……)、MacOS和Linux。文件是位于硬盘(计算机在关电后保存信息的部分)上的一组值(字节)的集合。如果你知道一个文件的名称并把它告诉操作系统,那么你就可以得到这个名字所关联的值。

你可能在想:“我使用计算机好多年了,从来没有向操作系统提供文件名字。”或许你在提供的时候没有意识到,实际上当你从Photoshop的文件选择对话框中选取一个文件,或者从目录窗口(或Explorer、Finder)中双击一个文件时,你都在要求某个软件将被选取或被双击的名字提供给操作系统并取回相关的值。然而,当你自己编写程序时,就要显式地获得文件名字并取得它们的值。

文件对媒体计算非常重要。磁盘上可以存储大片大片的信息。还记得我们讨论过的摩尔定律吗?每一元钱能买到的磁盘容量比每一元钱能买到运算速度增长得还要快!今天的计算机磁盘可以存储整部电影、数小时(或数日?)的声音,以及与几百卷胶卷等量的图片。

这些媒体的体积并不小,即使以压缩形式存放,一张屏幕大小的图片也会超过百万字节,一首歌曲可能有300万字节甚至更多。你需要把它们存放在计算机关电后仍然存在而且有大量空间的地方。

与磁盘不同,计算机的内存是不持久的(其内容会随着电源的中断而消失),而且空间相对较少。内存一直在变大,但与磁盘空间相比仍然是小巫见大巫。使用媒体时,你需要将它们从磁盘加载到内存,但工作完成之后你不会愿意把它们留在内存中。它们太大了。

可以把计算机的内存想象成一间宿舍。你可以方便地取用宿舍里的东西——它们就在你旁

边，伸手即得，用起来也方便。但你不会把自己拥有的一切（或者你希望拥有的一切）都放在那一间宿舍里。你所有的财产？你的滑雪板？你的汽车？你的小船？那太傻了。相反，你把大东西放在存放大东西的地方。你知道需要时如何得到它们（而且在你需要或条件允许时可将它们放回宿舍）。

你把信息带进内存时，就需要为数值命名，以便于之后取用它。从这个意义上，编程有点儿像代数（algebra）。为编写出通用（即对任意数字或数值都成立）的方程或函数，你会使用变量：就像 $PV = nRT$ ， $e = Mc^2$ 或者 $f(x) = \sin(x)$ 。这些 P 、 V 、 R 、 T 、 e 、 M 、 c 和 x 就是值的名字。当你计算 $f(30)$ 的值，知道自己在计算 f 且 x 是“30”的名字。在程序中使用媒体时，我们将采用（与数值）相同的方式来命名它们。

2.2 Python编程

本书将使用的语言称为Python。它是由Guido van Rossum发明的一门语言。Guido以著名的英国喜剧剧团Monty Python来命名自己的语言。许多未经正规计算机科学训练的人已经使用Python多年——Python的设计目标就是简单易用。本书将要使用的具体实现是Jython，因为它可以实现跨平台的多媒体编程。

实际上，你将使用一种称为JES（Jython Environment for Students）的工具来编写程序。JES是个简单的编辑器（输入程序文本的工具）和交互式工具，你可以在JES中尝试新事物，或者在其中创建新程序。本书所讨论的与媒体相关的名字（函数、变量、编码）将在JES中运行（也就是说，它们不是标准Python发布包的一部分，但我们使用的基础语言是标准Python）。

可以到<http://mediacomputation.org>上阅读JES的安装说明。那上面描述的过程将指导你安装Java、Jython和JES。安装之后，计算机上会有个漂亮的图标，双击它就能启动JES。JES有分别面向Windows、Macintosh和Linux的版本。



调试技巧：需要时别忘了安装Java

对于大多数人来说，只需将JES的文件夹拖到硬盘上就可以使用了。不过，如果你已经安装了Java，而且是个不能运行JES的老版本，那么启动JES时还是会有问题。倘若确实有问题，可以到Sun的网站<http://www.java.sun.com>上获取一份新版本的Java[⊖]。

2.3 JES编程

如何启动JES取决于你的平台。在Windows和Macintosh上，你会看到一个JES图标，双击它就能启动JES。在Linux上，你可能需要在Jython目录下输入一条命令，比如`./JES.sh`。你可以到网站上查阅相关说明，以确定自己的计算机应采用哪种启动方法。



常见bug：JES启动起来可能较慢

JES可能需要较长时间来加载。不必担心——你会看到长时间的启动画面，但只要你看到了启动画面，它就会加载。通常在第一次使用之后它会启动得更快。

⊖ 如今的新链接是：<http://www.oracle.com/technetwork/java/index.html>。——译者注



常见bug：让JES运行得更快

运行JES实际是在运行Java，这一点后面还会详细讨论。如果发现JES运行得有些慢，那就给它更多内存。你可以退出其他程序，从而达到这一效果。邮件程序、即时通信软件和数字音乐播放器都会占用内存——有时占用得还不少呢！退出这些程序后，JES就能运行得快一些。

启动之后的JES就如图2.1所示，其中有两块主要的区域（它们之间的分隔条可以移动，以便重新调整两块区域的大小）：

- 上面的部分是程序区。这是你编写程序（创建的程序和它们的名字）的地方。这一区域只是个文本编辑器——可以理解成用于编程的Microsoft Word。在你按下Load Program按钮之前，计算机不会真正解释你在程序区中输入的名字，而且，在程序保存之前你也按不了Load Program按钮（保存程序可以使用File菜单下的Save菜单项）。

不必担心按下了Load Program却忘记了保存程序。JES在程序保存之前不会加载它，会提供机会让你保存的。

- 底下的部分是命令区。这里是让你一句一句地命令计算机做事的地方。在“>>>”提示符之后输入命令，然后当按下<Return>（Apple）或<Enter>（Windows）键的时候，计算机就会解释你输入的词句（即对这些词句应用Python编程语言的含义和编码）并执行你让它做的事情。解释的内容还会包含你在程序区输入和加载的内容。

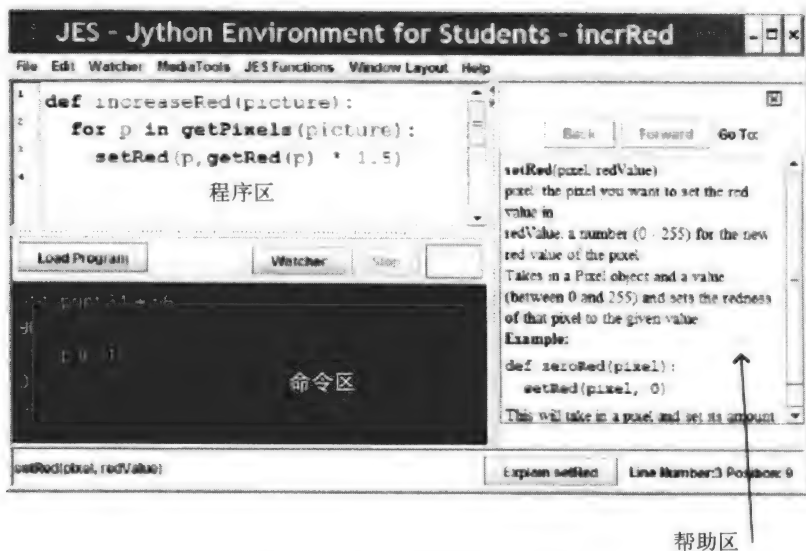
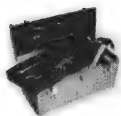


图2.1 标注了不同区域的JES

- 右边的区域是帮助区。你可以选择一项内容，然后点击相应的“Explain setRed”按钮来获得帮助。

图2.1中还可以看到JES的其他特性。但目前我们还不想用它们做太多事情。Watcher按钮可以打开一个观察器（调试器），观察器是一个窗口，通过里面的工具可以观察计算机怎样执行你的程序。Stop按钮允许你终止运行中的程序（比如你觉得它运行的时间太长，或者意识到它并没有做你想让它做的事情）。



实践技巧：了解你的助手

首先需要研究的一项重要特性就是Help菜单。这个菜单下面有丰富的帮助信息，都是关于编程和使用JES的。现在就开始研究它吧，这样当你开始编写自己的程序时，就大致知道那里有什么信息了。

2.4 JES媒体计算

我们将从在命令区输入命令的简单任务开始——暂时不定义新的名字，而只在JES中使用计算机已经知道的名字。

`print`是我们需要知道的一个重要名字。使用时，它后面总是跟着其他东西。`print`的含义是：“以一种可读的形式显示后面的东西，不管它是什么。”后面跟着的可能是个计算机知道的名字，也可能是个表达式（`expression`）（与代数学意义上的表达式差不多）。试着点击命令区并输入命令：`print 34 + 56`，然后输入Enter键——就像这样：

```
>>> print 34 + 56
90
```

`34 + 56`是Python所能理解的一个数字表达式。很明显，它由两个数字和一种Python知道如何完成的操作（就是我们所说的名字）组成。“+”的意思是“加”。Python还理解其他类型的表达式，不一定是数字表达式。

```
>>> print 34.1/46.5
0.7333333333333334
>>> print 22 * 33
726
>>> print 14 - 15
-1
>>> print "Hello"
Hello
>>> print "Hello" + "Mark"
HelloMark
```

Python理解很多标准数学操作。它也知道如何识别不同种类或类型的数字：整数和浮点数。浮点数有小数点，整数没有小数点。Python还知道如何识别以双引号（"）开始和结束的字符串（字符序列）。它甚至知道如何将两个字符串“相加”：无非是把一个字符串放到另一个之后。

常见bug：Python的类型会产生奇特的结果



Python对待类型是严肃的。如果它看到你使用整数，那么它认为你想从表达式中得出一个整数结果。如果它看到你使用浮点数，那么它认为你想得到浮点数结果。听上去很合理，不是吗？那下面这种情况呢？

```
>>> print 1.0/2.0
0.5
>>> print 1/2
0
```

`1/2`结果为0？嗯，当然是的！1和2是整数。没有等于`1/2`的整数，因此结果只能是0！给整数加上一个“.”就能向Python表明我们讨论的是浮点数，于是结果也就成了浮点数形式。

Python还理解函数 (function)。还记得代数中讲的函数吗？它们是一种“盒子”，放进去一个值，出来另一个值。Python认识的函数当中有一个接收单个字符作输入值（放进盒子的值），返回或输出（从盒子里出来的值）一个数字，该数字是那个输入字符的ASCII映射码。这个函数的名字叫ord (ordinal)，你可以用print显示ord函数返回的值：

```
>>> print ord("A")
65
```

另一个Python内置的函数叫abs——绝对值函数，它返回输入值的绝对值：

```
>>> print abs(1)
1
>>> print abs(-1)
1
```



调试技巧：常见的拼写错误

如果你输入了Python根本不理解的东西，那么就会得到一条语法错误提示。

```
>>> pint "Hello"
Your code contains at least one syntax
error, meaning it is not legal Jython.
```

如果你试图访问一个Python不知道的词语，Python会说它不认识那个名字。

```
>>> print a
A local or global name could not be found. You need to define
the function or variable before you try to use it in any way.
```

局部名字 (local name) 是函数内部定义的名字，全局名字 (global name) 是所有函数都能访问的名字 (如pickAFile)。

JES认识的另一个函数允许你从磁盘上选择文件。你可能注意到了，不再说“Python认识”，而改说“JES认识”。print是所有Python实现都认识的东西，pickAFile却是为JES开发的。通常，你可以忽略这种区别，但如果你要尝试使用另一种Python，了解哪些部分是通用的、哪些部分不是通用的就很重要了。与ord不同，这个函数不需要输入，但会返回一个字符串，该字符串是你磁盘上某个文件的名称。这个函数的名字叫pickAFile。Python对大小写非常挑剔——写成pickafile或者Pickafile都不正确。试着用一下print pickAFile()，运行时你会看到图2.2所示的窗口。

关于如何使用文件选择器或者文件对话框，你应该很熟悉了：

- 通过双击文件夹/目录打开它们。
- 通过点击选择文件，然后点击Open按钮；也可以直接双击文件。

选择了文件之后，pickAFile返回一个字符串（一系列字符）作为全路径文件名（full filename）。（如果单击了Cancel，pickAFile就返回空字符串——一个不包含任何字符的字符串，比如"）。尝试一下：运行print pickAFile()并Open一个文件。

```
>>> print pickAFile()
C:\ip-book\mediasources\beach.jpg
```

选择文件时最终得到的字符串形式与你使用的操作系统有关。在Windows中，文件名可能以C:开头且含有反斜杠 (\)。在Linux或MacOS中则可能是这个样子：/Users/guzdial/ip-

book/mediasources/beach.jpg。这个文件名实际包含了两部分：

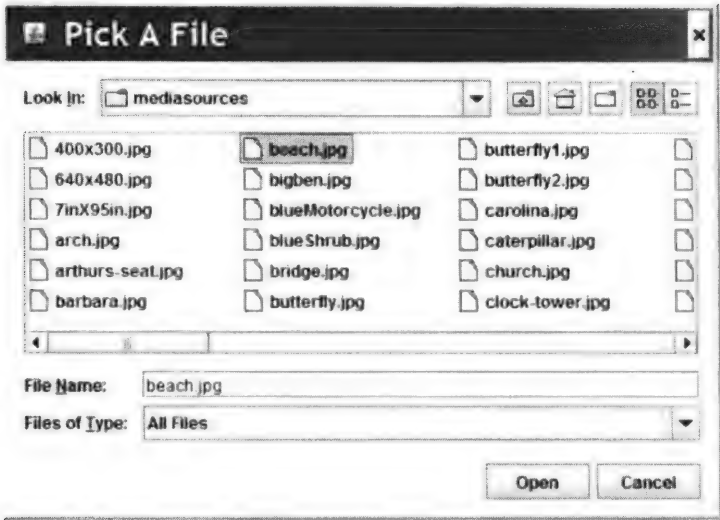


图2.2 文件选择器

- 各单词之间的字符（比如“Users”和“guzdial”之间的/）称为路径分隔符。从文件名开头到最后一个路径分隔符之间的所有部分称为到达文件的路径。它精确地描述了文件存在于磁盘上的位置（即存在于哪个目录中）。
- 文件名的最后一部分（如“beach.jpg”）叫做**基本文件名**（base filename）。当你在Finder、Explorer或Directory窗口（取决于你的操作系统）中查看文件时，看到的就是这一部分。最后三个字符（.之后的）叫做**文件扩展名**（file extension）。它们标识了文件使用的编码。扩展名为“.jpg”的文件是JPEG文件。这些文件包含图片内容。（更严格地讲，它们包含可以按图片的表示格式来解释的数据——不过，说它们“包含图片内容”也很接近这个意思了。）JPEG是一种标准编码（表示），可用于编码各种图像。我们还将经常使用的另外一种文件是“.wav”文件（如图2.3所示）。“.wav”扩展名表明它们是WAV文件。它们包含声音内容。

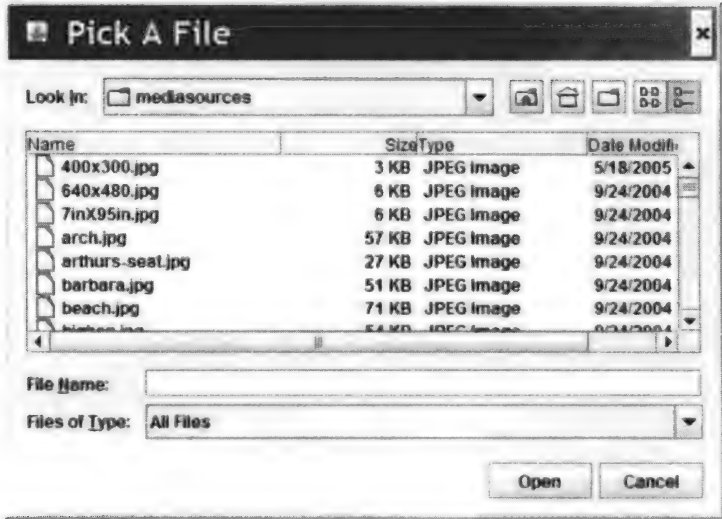


图2.3 标识出媒体类型的文件选择器

WAV是声音的标准编码。还有很多其他类型的文件扩展名，甚至许多其他类型的媒体文件扩展名。举例来说，还有表示图像的GIF（“.gif”）文件和表示声音的AIFF（“.aif”或“.aiff”）文件。为简单起见，本书只考察JPEG和WAV。

2.4.1 显示图片

现在，我们知道了如何获取一个完整的文件名：路径和基本名。但这不表明我们把文件本身加载到内存中了。为了把文件加载到内存中，我们必须告诉JES如何解释它。我们知道JPEG文件是图片，但我们必须显式地告诉JES读入文件并从中构造一幅图片。针对这一功能也有一个函数，叫做makePicture。

makePicture也需要一个参数（parameter）——即函数的输入。与ord一样，在括号中指定函数的输入。makePicture函数接受一个文件名。运气不错——我们知道如何获取一个文件名。

```
>>> print makePicture(pickAFile())
Picture, filename C:\ip-book\mediasources\barbara.jpg
      height 294 width 222
```

print函数的结果显示：基于指定的文件名，以及高度、宽度，我们确实构造了一幅图片。成功！哦，你想直接看到真正的图片？那我们需要另一个函数（计算机很蠢的，我们有没有在别处提过？）显示图片的函数叫show。show同样接受一个参数——一个Picture。

但现在我们有问题了。我们没有为刚刚创建的图片取个名字，因此无法再次引用它。我们不可说“显示一秒钟前创建但没有命名的那幅图片”。除非我们给它命名，否则计算机根本记不住任何东西。为一个值创建名字的过程也称为声明变量。

让我们重新来一次，这一次首先为选取的文件命名。我们还会为创建的图片命名，然后我们就能显示命名的图片，如图2.4所示。

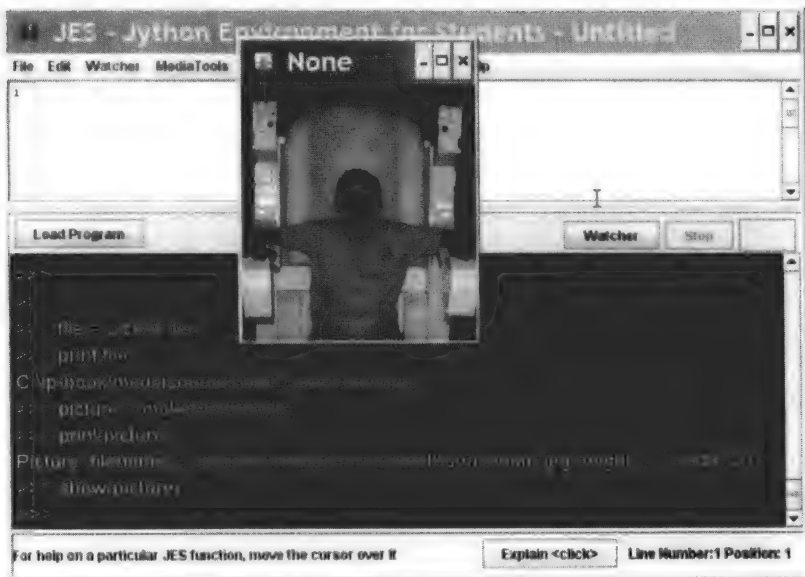


图2.4 选取、构造并显示一幅图片，所有的值都被命名

你可以使用 `file = pickAFile()` 来选择并命名一个文件。其含义是创建一个有名字的文件并使之引用 `pickAFile()` 函数返回的值。我们已经知道 `pickAFile()` 函数能返回文件的名称（包括路径）。我们可以使用 `pict = makePicture(file)` 来创建一幅图片并为之命名，它把文件名传给 `makePicture` 函数，并返回创建的图片。名字 `pict` 引用创建的图片。然后我们可以把名字 `pict` 传给函数 `show`，以此来显示创建出来的图片。

另外一种方法是用一个函数完成所有动作，因为一个函数的输出可以作为另一个函数的输入：`show(makePicture(pickAFile()))`。在图2.5中可以看到这种用法。它让你从选择一个文件开始，把文件的名称传给 `makePicture` 函数；然后再把结果图片传给 `show` 函数。但这次我们又没给图片命名，因此无法再次引用它了。



图2.5 选取、构造并显示一幅图片，每个函数直接用做下一个函数的输入

自己动手把两种方法都试一下吧。恭喜！你已经完成了第一次媒体计算！

倘若试一下 `print show(pict)`，你会发现 `show` 输出的是 `None`。与实际的数学函数不同，Python 中的函数不一定要有返回值。只要函数能完成一些功能（比如在窗口中打开一幅图片），即便没有返回值，它也是有用的。计算机科学家使用术语副作用（side-effect）来描述一个函数完成其他计算，而不是从输入到返回值计算的情况。

2.4.2 播放声音

我们可以用声音媒体来重复整个过程：

- 仍然用 `pickAFile` 找到想要的文件并取得文件名。这次选取的是以 `.wav` 结尾的文件。
- 这次用 `makeSound` 来构造声音。如你所想，`makeSound` 接受一个文件名作为输入。
- 将使用 `play` 来播放声音。`play` 接收声音值作为输入，返回 `None`。

以下的步骤与我们前面显示图片时是一致的：

```
>>> file = pickAFile()
>>> print file
C:\ip-book\mediasources\hello.wav
>>> sound = makeSound(file)
```

```
>>> print sound

Sound of length 54757
>>> print play(sound)
None
```

(我们将在下一章解释声音长度的含义。)请运行一下这些命令，使用自己的计算机上自己制作的，或者从<http://www.mediacomputation.org>下载的JPEG文件和WAV文件。(关于到哪里获取媒体，以及如何创建它们，后续章节会有更多讨论。)

2.4.3 数值命名

从上一节可以看到，使用“=”来命名数据。我们可以用print来检查命名，正如前面做的那样。

```
>>> myVariable=12
>>> print myVariable
12
>>> anotherVariable=34.5
>>> print anotherVariable
34.5
>>> myName="Mark"
>>> print myName
Mark
```

不要把“=”读作“等于”，那是它在数学中的含义。在这里，我们用的不是这个含义。应该把它读作“成为……的名字”。于是，myVariable = 12的意思是：“myVariable成为12的名字”。因此反过来写（表达式放左边，名字放右边）是无意义的：这样一来，12 = myVariable岂不成了“12成为myVariable的名字”了。

```
>>> x = 2 * 8
>>> print x
16
>>> 2 * 8 = x
Your code contains at least one syntax error, meaning
it is not legal Jython.
```

可以多次使用一个名字。

```
>>> print myVariable
12
>>> myVariable="Hello"
>>> print myVariable
Hello
```

名字和相应数据之间的绑定（或关联）仅在以下情况之前存在：(a) 名字被赋予其他数据；(b) 退出JES。名字和数据（甚至名字和函数）之间的关系仅存在于一次JES会话(session)中。

记住：数据拥有编码和类型。数据在表达式中的行为方式部分地取决于它的类型。留意一下整数12和字符串“12”在下面的操作中会有怎样的不同。针对其类型来说，它们都完成了合理的动作，但却是截然不同的动作。

```
>>> myVariable=12
>>> print myVariable*4
```



```

48
>>> myOtherVariable="12"
>>> print myOtherVariable*4
12121212

```

也可以把名字赋予函数执行的结果。如果为pickAFile函数命名结果,那么每次打印这个名字时都将得到同样的结果。但没有重新执行pickAFile。另外,为代码命名以便重新执行它是定义函数时所做的事情,几页之后便会讲到。

```

>>> file = pickAFile()
>>> print file
C:\ip-book\mediasources\640x480.jpg
>>> print file
C:\ip-book\mediasources\640x480.jpg

```

在下面的例子中,把名字赋予文件名和图片。

```

>>> myFilename = pickAFile()
>>> print myFilename
C:\ip-book\mediasources\barbara.jpg
>>> myPicture = makePicture(myFilename)
>>> print myPicture
Picture, filename barbara.jpg
height 294 width 222

```

注意,代数的替换(substitution)和求值(evaluation)概念在这里同样有效。myPicture = makePicture(myFilename)与makePicture(pickAFile())创建的图片是完全一样的[⊖],因为我们让myFilename等于pickAFile()的结果。名字在表达式求值的时候被替换为相应的值。makePicture(myFilename)表达式求值时被展开为makePicture("C:/ip-book/mediasources/barbara.jpg"),因为"C:/ip-book/mediasources/barbara.jpg"是pickAFile()求值时我们选取的文件,而返回值被命名为myFilename。

我们还可以用函数返回的值来替换函数调用(invocation或call)而保持结果不变。pickAFile返回一个字符串——双引号括起来的一串字符。我们可以把上一个例子改写成下面这样,效果仍然不变。

```

>>> myFilename = "C:/ip-book/mediasources/barbara.jpg"
>>> print myFilename
C:/ip-book/mediasources/barbara.jpg
>>> myPicture = makePicture(myFilename)
>>> print myPicture
Picture, filename C:/ip-book/mediasources/barbara.jpg
height 294 width 222

```

甚至替换掉名字。

```

>>> myPicture = makePicture("C:/ip-book/mediasources/
    barbara.jpg")
>>> print myPicture
Picture, filename C:/ip-book/mediasources/barbara.jpg
height 294 width 222

```

⊖ 当然,假定选取的是同一个文件。

计算机科学思想：名字、值和函数可以相互替换



值、赋予该值的名字和返回相同值的函数可以相互替换。计算机关心的是值，而不关心它究竟来自字符串、名字还是函数调用。关键在于计算机要对值、名字和函数求值（evaluate）。只要这些表达式可以求出相同的字符串，那么它们就可以相互替换。

事实上，我们不需要每次让计算机做点什么的时候都使用print。如果调用一个不返回任何东西的函数（这样的函数也print不出有用的东西），那么我们可以输入函数的名字及其输入（如果有的话）来调用它，然后直接按Enter键。

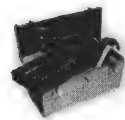
```
>>> show(myPicture)
```

这些让计算机做事情的语句，我们常常称之为命令。print myPicture就是一条命令。myFilename = pickAFile()和show(myPicture)也是命令。它们不仅是表达式：它们让计算机做事情。

2.5 构建程序

我们已经会用名字来表示值。表达式求值的时候名字被替换成相应的值。对程序也可以做同样的事情。我们可以给一系列命令取个名字，然后每次想执行这些命令时使用这个名字即可。在Python中，我们为命令定义的名字将是一个函数。于是，Python中的程序就是由一个或多个执行有用任务的函数组成的集合。我们仍将使用术语菜谱（recipe）来描述执行有用媒体计算的程序（或程序的某些部分），虽然有时候它所涵盖的东西本身不足以成为有用程序。

还记得我们之前说过的吗？计算机上几乎所有的东西都可以命名。我们已经见过了为数值命名，现在看一下为菜谱命名。



实践技巧：尝试每一份菜谱

要真正理解每一份菜谱做了什么，就应该输入、加载并执行书中的每一份菜谱。强调一下，是每一份。它们都不长，但在确信程序能正常工作、开发编程技能和理解程序为什么能够工作等方面，这些实践大有裨益。

针对定义新菜谱，Python所理解的名字是def。def不是函数，它是一个命令，就像print一样。def用于定义新的函数。不过，单词def之后还必须跟上特定的内容。与def命令相关的内容结构称为命令的语法（syntax），为了让Python理解这里是什么以及这些内容的次序而必须出现的单词和字符。

def之后需要在同一行上有三样东西：

- 要定义的菜谱名字，比如showMyPicture。
- 菜谱接受的所有输入。菜谱可以是接受输入的函数，像abs和makePicture。各个输入要有名字，且置于括号中以逗号（,）分隔。如果没有输入，只需输入一对括号“()”来表示。
- 整行定义以冒号（:）结尾。

def行之后便是每次执行菜谱时将会执行的命令，一条接着一。通过定义命令块，我们可以创建由一组命令组成的集合。函数的名字所关联的命令就包含在def命令（或语句）之后的命令块中。

大多数完成有用功能的实际程序，特别是那些创建用户界面的程序，都需要定义多个函数。想象一下程序区中有多条def命令的情况。你觉得Python会如何确定上一个函数的结束和下一个函数的开始呢？（特别要考虑函数内部可以定义其他函数的可能。）Python需要一种确定函数体（function body）结束的方法，即确定哪些语句是这个函数的一部分，哪些是下一个函数的一部分。

答案是缩进（indentation）。属于一个函数定义的所有语句相对于def语句都要缩进一点。我们建议使用不多不少两个空格来缩进——足以看清，便于记忆且简单易行。（在JES中，你还可以使用跳格键（按一次Tab键）来缩进。）你可以像下面这样在程序区输入函数（这里的“”表示单个空格，即按一次空格键输入的字符）：

```
def hello():
    print "Hello"
```

现在，我们可以定义自己的第一份菜谱了。可以把下面的程序输入JES程序区。输完之后保存文件，使用扩展名“.py”来表示这是个Python文件。（我们自己使用的名字是pickAndShow.py。）



程序1：选取并显示图片

```
def pickAndShow():
    myFile = pickAFile()
    myPict = makePicture(myFile)
    show(myPict)
```

输入程序的时候，你会注意到函数体的四周出现的一个浅蓝色框。这个蓝色框显示的就是程序中的块（如图2.6所示）。与包含光标（接受输入处的竖杠）的语句处于同一块中的所有命令都圈在同一个蓝框中。如果你希望处于同一块的所有命令都出现在同一个蓝框中，那么你能知道自己的缩进是正确的。

输入完菜谱并保存之后，你就可以加载它了。点击Load Program按钮。

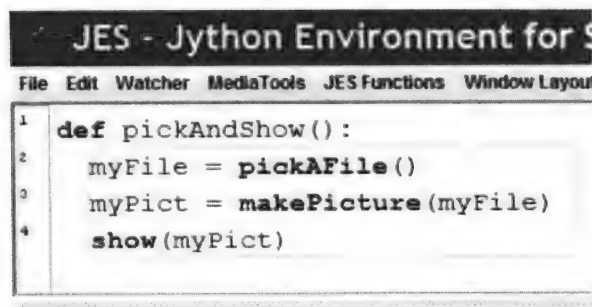


图2.6 JES中的可视化命令块

调试技巧：别忘了加载

使用JES时，最常见的错误就是输入并保存了函数，然后还没有加载它就在命令区尝试使用函数。你需要点击Load Program按钮把函数加载进来，这样命令区才能使用它。

现在你可以执行自己的菜谱。点击一下命令区。你的菜谱既不接受输入也不返回值（也就是说，这不是严格数学意义上的函数），所以只需要把它的名字作为一条命令输入就可以了：

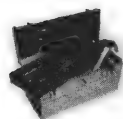
```
>>> pickAndShow()
>>>
```

可以用类似的方法定义第二份菜谱，选取并播放一段声音。



程序2：选取并播放声音

```
def pickAndPlay():
    myFile = pickAFile()
    mySound = makeSound(myFile)
    play(mySound)
```



实践技巧：选用自己喜欢的名字

在上一节中，我们使用了名字myFilename和myPicture。这份菜谱中，我们使用了myFile和myPict。这有关系吗？对计算机来说无所谓。我们可以把图片命名为myGlyph甚至myThing。计算机并不关心你使用什么名字——名字完全是为你服务的。你应该选用这样的名字：(a) 对你有意义（这样你可以阅读并理解自己的程序）；(b) 对别人有意义（这样当你把程序展示给别人时，别人也能理解）；(c) 易于输入。像myPictureThatIAmGoingToOpenAfterThis这种包含30多个字符的名字虽然有意义且易于阅读，但输入的时候太痛苦了。

作为程序而言，这些菜谱可能没什么实际用处。如果你想显示同一幅图片，那么一遍遍地选取文件很烦人。既然有定义菜谱的能力，那么我们就可以定义新的菜谱来执行自己想要的任何动作。让我们定义一个打开特定图片的菜谱和另一个打开特定声音的菜谱。

使用pickAFile来获得你想要的声音或图片的文件名。我们在定义播放那段声音或显示那幅图片的菜谱时，将需要这个文件名。在这个菜谱中，我们直接把文件名字符串用双引号引起来，然后直接设置myFile的值，而不是使用pickAFile的结果。



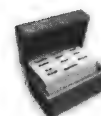
程序3：显示特定图片

别忘了使用你自己图片文件的完整路径名替换下面程序中的FILENAME。比如“C:/ip-book/mediasources/barbara.jpg”。

```
def showPicture():
    myFile = "FILENAME"
    myPict = makePicture(myFile)
    show(myPict)
```

程序原理

变量myFile接受了文件名的值——它与pickAFile函数返回的是同一个值（如果你选择那个文件的话）。然后，我们基于这个文件构造了一幅图片并命名为myPict。最后，我们显示了myPict中的图片。



程序4：播放特定声音

别忘了使用你自己声音文件的完整路径名替换下面程序中的FILENAME。比如“C:/ip-book/mediasources/hello.wav”。

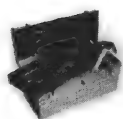
```
def playSound():
    myFile = "FILENAME"
    mySound = makeSound(myFile)
    play(mySound)
```



常见bug: Windows文件名和反斜杠

Windows使用反斜杠（'\'）作为文件名分隔符。Python为某些反斜杠-字符的组合赋予了特殊含义，这一点我们后面会讲到更多。比如，'\n'的意思与Enter或Return键的输入是一样的。这些组合有可能自然地出现在Windows的文件名中。为避免Python错误地解读这些字符，你可以改用正斜杠（'/'），就像“C:/ip-book/mediasources/barbara.jpg”这样，或者你可以在文件名前输入一个“r”，就像：

```
>>> myFile = r"C:\ip-book\mediasources\barbara.jpg"
>>> print myFile
C:\\ip-book\\mediasources\\barbara.jpg
```



实践技巧: 复制和粘贴

可以在程序区和命令区之间复制和粘贴文本。你可以用print pickAFile()来打印一个文件名，然后选中并复制（Copy，从Edit菜单中）它。然后点击命令区再粘贴（Paste）它。类似地，你可以从命令区把完整的命令复制到上面的程序区。这是一种非常便利的测试程序的方法：先测试各条命令，当次序和结果都正确的时候再放在一起做成一份菜谱。你还可以在命令区内部复制文本。选中一条命令，复制它，将它粘贴到最后一行（确定光标位于行末！），然后输入Enter键执行就可以了，不必每次运行同样的命令时都重新输入一遍。

可变菜谱: 真正接受输入的一类数学函数

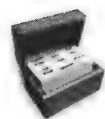
如何创建一个真正的函数呢？就像数学中接受输入的函数那样，比如ord和makePicture。我们又因何需要这样的函数呢？

使用变量来指定菜谱输入的一个重要原因是让程序更加通用。考虑程序3: showPicture。该程序针对一个特定的文件名。如果能有一个能接受任意文件名并从中构造、显示图片的函数会不会更有用呢？这种函数可以处理构造并显示图片的一般情形。我们将这种一般化的过程称为抽象（abstraction）。抽象可以带来适应多种情形的一般解决方案。

定义一个接受输入的菜谱非常容易。这依然是个替换与求值的问题。我们在def行的括号中放入一个名字。这个名字有时称为形参（parameter）或输入变量。

当你指定函数的名字，并在括号中给出输入值，也称为实参（argument），就像makePicture(myFilename)或show(myPicture)，在函数求值的时候，输入值就被赋值给输入变量。我们说输入变量接受（take on）了输入值。在函数（菜谱）执行期间，输入值将替换这个输入变量。

下面就是一个接受文件名作为输入变量的菜谱：



程序5: 显示图片文件，文件的名称作为输入

```
def showNamed(myFile):
    myPict = makePicture(myFile)
    show(myPict)
```

点击Load Program按钮，让JES把函数读入程序区。如果函数中有错误，那么你需要改正它们，然后重新点击Load Program按钮。一旦成功加载了函数，那么你就可以在命令区使用它们。

当你在命令区输入：

```
showNamed("C:/ip-book/mediasources/barbara.jpg")
```

并按Enter（回车）键，showNamed函数中的变量myFile就接收了该值：

```
"C:/ip-book/mediasources/barbara.jpg"
```

然后，myPict变量引用了读取并解释文件所得的结果，然后图片就显示出来了。

可以用相同的方法创建一个播放声音的函数。我们可以在程序区输入多个函数。试着在前面的函数之后增加如下函数，然后再次点击Load Program按钮。



程序6：播放声音文件，文件的名字作为输入

```
def playNamed(myFile):
    mySound = makeSound(myFile)
    play(mySound)
```

可在命令区输入以下命令来试用这个函数。

```
>>> playNamed("C:/ip-book/mediasources/croak.wav")
```

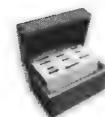
也可以创建接受多个参数的函数，用逗号把各参数隔开即可。



程序7：播放声音文件的同时显示图片

```
def playAndShow(sFile, pFile):
    mySound = makeSound(sFile)
    myPict = makePicture(pFile)
    play(mySound)
    show(myPict)
```

还可以编写接收图片或声音作为输入值的程序。下面是一个显示图片的程序，但它接收图片对象作为输入，而不是文件名。



程序8：显示作为输入传入的图片

```
def showPicture(myPict):
    show(myPict)
```

这时你可能想把这些函数存入文件以便再次使用它们。点击File，然后点击Save Program。这时会有一个文件对话框显示出来，你可以指定文件的名字和存放位置。之后如果你退出并重新启动了JES，那么你就可以用File菜单中的Open Program重新打开文件，并点击Load Program来加载函数以供使用。

showPicture函数与JES内置的show函数有什么区别呢？没有任何区别。我们当然可以创建一个函数来给另一个函数提供新名称。如果那样更便于你理解自己的代码，那么它就是个好主意。

对函数来说，恰当的输入值应该是什么呢？输入文件名更好还是输入图片对象更好？而这里的“更好”又是什么意思呢？关于这些问题后面会有更多讨论。但这里先给一个简单答案：编写对自己最有用的函数。如果对你来说定义showPicture比使用show的可读性更好，那么这

样的定义就是有用的。如果你真正想要的是一个全权负责把图片构造出来并显示的函数，那么你可能觉得showNamed是最好用的。

编程摘要

本章讨论了以下几种数据（或对象）的编码：

整数（如：3）	没有小数点的数字——不能表示分数
浮点数（如：3.0、3.01）	可包含小数点的数字——可以表示分数
字符串（如："Hello!"）	一系列字符（包括空格、标点符号等），两端各有一个双引号
文件名	一个字符串，其中的字符表示了一个路径和一个基本文件名
图片	图像的编码，通常来自JPEG文件
声音	声音的编码，通常来自WAV文件

以下是本章介绍的程序片段：

print	以文本形式显示表达式（变量、值、算式等）的值
def	定义函数及其输入变量（如果有的话）
ord	返回与输入字符等价的数字值（根据ASCII标准）
abs	接受一个数字并返回其绝对值
pickAFile	让用户选取一个文件，并以字符串形式返回其完整文件名
makePicture	接受路径名作为输入，读入文件并基于文件内容构造图片，返回新构造的图片对象
show	显示作为输入传入的图片。不返回任何东西
makeSound	接受路径名作为输入，读入文件并基于文件内容构造声音，返回新构造的声音对象
play	接受声音对象并播放它。不返回任何东西

习题

2.1 计算机科学概念问题：

- 什么是算法？
- 什么是编码？
- 计算机如何以数字形式表示图片？
- 摩尔定律是什么？

2.2 def的含义是什么？语句def someFunction(x, y):做什么事情？

2.3 print的含义是什么？语句print a做什么事情？

2.4 print 1 / 3会输出什么结果？为什么会输出这样的结果？

2.5 print 1.3 * 3会输出什么结果？为什么会输出这样的结果？

2.6 print 1.0 / 3会输出什么结果？为什么会输出这样的结果？

2.7 print 10 + 3 * 7会输出什么结果？为什么会输出这样的结果？

2.8 print (10 + 3) * 7会输出什么结果？为什么会输出这样的结果？

2.9 print "Hi" + "there"会输出什么结果？为什么会输出这样的结果？

2.10 以下命令的输出是什么？

```
>>> a = 3
>>> b = 4
>>> x = a * b
>>> print x
```

2.11 以下命令的输出是什么?

```
>>> a = 3
>>> b = -5
>>> x = a * b
>>> print x
```

2.12 以下命令的输出是什么?

```
>>> a = 4
>>> b = 2
>>> x = a / b
>>> print x
```

2.13 以下命令的输出是什么?

```
>>> a = 4
>>> b = 2
>>> x = b - a
>>> print x
```

2.14 以下命令的输出是什么?

```
>>> a = -4
>>> b = 2
>>> c = abs(a)
>>> x = a * c
>>> print x
```

2.15 以下命令的输出是什么?

```
>>> name = "Barb"
>>> name = "Mark"
>>> print name
```

2.16 以下命令的输出是什么?

```
>>> a = ord("A")
>>> b = 2
>>> x = a * b
>>> print x
```

2.17 下面的代码导致了它后面的错误消息, 试着改正它。

```
>>> pickafile()
The error was:pickafile
Name not found globally.
A local or global name could not be foun . You need
to define the function or variable before you try
to use it in any way.
```

2.18 下面的代码导致了它后面的错误消息, 试着改正它。

```
>>> a = 3
>>> b = 4
>>> c = d * a
The error was:d
```

```
Name not found globally.
A local or global name could not be found. You need
to define the function or variable before you try
to use it in any way.
```

- 2.19 `show(p)`是做什么的？（提示：该问题的答案不止一个。）
- 2.20 试着在JES中使用其他一些字符串操作。如果把数字与字符串相乘，比如`3 * "Hello"`，结果会怎样？字符串乘以字符串，比如`"a" * "b"`，结果又会怎样？
- 2.21 当我们要执行名为`pickAFile`的函数时，会对表达式`pickAFile()`求值。那名字`pickAFile`本身又是什么呢？如果执行`print pickAFile`，会得到什么结果？执行`print makePicture`呢？打印出的内容是什么？你又是如何理解其含义的呢？

深入学习

最好（最深入、最具体、最优雅）的计算机科学教程是Abelson、Sussman和Sussman（这里不是重复，一个是Gerald Jay Sussman，另一个是Julie Sussman。参阅书后的“参考书目”。——译者注）合著的《Structure and Interpretation of Computer Programs》（中文版《计算机程序的构造和解释》，机械工业出版社。——译者注）[2]。读完这本书是一项颇具挑战性的任务，但绝对值得。另一本较新的书《How to Design Programs》（中文版《程序设计方法》，人民邮电出版社。——译者注）[12]更注重面向编程新手的内容，但主体思想与上一本是一致的。

然而，这两本书针对的都不是因为兴趣而编程或者因为要做一些小事而编程的学生。它们针对的都是未来的专业软件开发人员。对探索计算机的学生来说，最好的书是Brian Harvey编写的，《Simply Scheme》，该书使用的编程语言与Abelson等人的书一样，但更加易懂。这类书中，我们最喜欢的却是Harvey的三卷本《Computer Science Logo Style》（这里的“Logo”指的是一门编程语言。——译者注）[23]，这本书把有益的计算机科学与富有创造性的有趣项目有机地结合了起来。

使用循环修改图片

本章学习目标

本章媒体学习目标：

- 理解如何利用人类的视觉限制把图片数字化。
- 识别不同的颜色模型，包括计算机中最常用的RGB。
- 处理图片中的颜色值，比如增加或减少红色值。
- 通过多种方法将彩色图片转换为灰度图片。
- 图片的颜色反转。

本章计算机科学学习目标：

- 使用矩阵表示查找图片中的像素。
- 使用图片对象和像素对象。
- 使用（基于for循环的）迭代改变图片中的像素颜色值。
- 代码块嵌套。
- 在有返回值的函数和仅提供副作用的函数之间做选择。
- 确定变量名的作用域。

3.1 图片的编码

图片（或图像）是所有媒体通信的重要组成部分。本章讨论图片在计算机上的表示方式（主要讲位图图像——单独表示每个点或像素），以及如何处理它们。下一章将介绍其他的图像表示法，比如向量图像（vector image）。

图片是二维的像素数组。这一节将描述所有这些术语。

就我们的目标而言，图片就是存储在JPEG文件中的图像。JPEG是关于如何以较少空间存储高品质图像的国际标准。JPEG是一种有损压缩（lossy compression）格式。这意味着它是压缩的，尺寸更小，但并不具有原始格式100%的品质。当然，通常舍弃掉的是你看不到或者注意不到的东西。就大多数应用而言，使用JPEG图像效果都很不错。

一维数组是类型相同的一系列元素。可以为数组命名，然后用下标来访问数组中的元素。数组中第一个元素的下标为0。在图3.1中，下标0处的值为15，下标1处的值为12，下标2处的值为13，下标3处的值为10。

二维数组也称为矩阵（matrix）。矩阵是以行和列的形式排列的一组元素的集合。这意味着要访问矩阵中的值可以同时指定行下标和列下标。

图3.2就是一个矩阵的例子。在坐标（0，1）（横，纵）处可以找到值为9的矩阵元素。（0，0）是15，（1，0）是12，（2，0）是13。我们会经常以（x，y）（横，纵）的形式来引用这些坐标。

0	1	2	3
15	12	13	10

图3.1 数组示例

	0	1	2	3
0	15	12	13	10
1	9	7	2	1
2	6	3	9	10

图3.2 矩阵示例

图片的每个元素中存储的是一个像素（pixel）。pixel是“picture element”的缩写。它实际上就是一个点，而整幅图片就是由大量这样的点组成的。你有没有试过拿一个放大镜看报纸或杂志上的图片，或者看电视甚至显示器？当你看着杂志或电视上的图片时，它似乎并没有被分解成几百万个离散小点，但实际上它是的。

使用图片工具可以得到类似的一个个像素的视觉效果，如图3.3所示。这一工具能让你将图片放大到500%，于是每个像素都是可见的。要使用图片工具，方法之一是像下面的程序这样浏览图片。

```
>>> file = "c:/ip-book/mediasources/caterpillar.jpg"
>>> pict = makePicture(file)
>>> explore(pict)
```

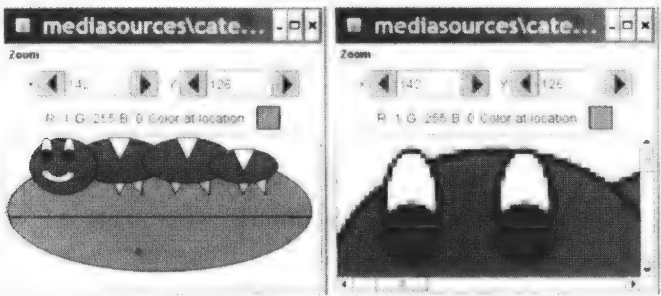


图3.3 显示在JES图片工具中的图像：左侧的显示比例是100%，右侧的显示比例是500%

我们人类的感官无法从事物的整体中分辨出极其细小的部分（除非凭借放大设备或其他特殊设备）。人类的视觉敏锐度（acuity）很低——比如说，我们看到的细节不如老鹰多。实际上，我们大脑和眼睛中的视觉系统不止一种。处理彩色的系统和处理黑白色（或者亮度）的系统是不一样的。比如，我们通过亮度来检测运动和物体的尺寸。实际上，眼睛的边缘部分比中心部分更适于拾取亮度。这是一种进化优势，它能让你提取到龇着利齿的老虎正从右边的灌木丛中悄悄地向你走来。

人类视觉分辨力的限制使图片的数字化成为可能。有些动物（比如，鹰和猫）能观察到比人类更多的细节，实际上，这些动物能看清图片上的单个像素。我们把图片分解成更小的元素（像素），但这些像素足够多且足够小，于是人们从整体上看时不会觉得断断续续。如果你能看清数字化的效果（比如，你可以看到某些位置的小矩形），那么我们把这种效果称为像素化（pixelization）——即数字化过程显而易见的效果。

图片编码比声音编码更加复杂。声音本来是线性的——它沿着时间一直向前。图片却有两个维度：宽度和高度。

可见光是连续的——可见光的波长在370~730纳米(0.000 000 37~0.000 000 73米)之间。但我们对光的感知却受到色感器官(color sensor)工作方式的限制。我们的眼睛拥有几个色感器官,它们分别在感受到波长为425纳米(蓝)、550纳米(绿)和560纳米(红)左右的光时被触发(达到峰值)。我们的大脑基于眼睛中这三种色感的反馈来确定一种颜色。有些动物(比如狗)只有两种色感。这些动物仍然能感知颜色。但看到的颜色或感知的方式与人类不一样。关于我们能力有限的视觉感官,一个有趣的结果是:我们实际上能感知两种橙色。一种是光谱橙色——自然橙色的特定波长的光。另一种是红、黄的某种混合,当它冲击我们的色感器官时,使我们恰好感知成相同的橙色。

根据人类感知颜色的方式,只要能把冲击三种颜色感官的光线进行编码,我们就能记录人类所感知的颜色。于是,我们把每个像素编码成三个数字。第一个数字表示像素中的红色分量,第二个是绿色分量,第三个是蓝色分量。通过组合红、绿和蓝色的光,我们能构造出所有人类能看到的颜色(如图3.4所示)。三种颜色的光全部组合起来可以得到纯白色。三种全部关闭则是黑色。我们把这称为RGB颜色模型。

除了RGB颜色模型之外,还有其他定义和编码颜色的模型。有一种叫HSV颜色模型,它编码了色调(Hue)、饱和度(Saturation)和颜色值(Value)(有时也称为HSB颜色模型,三个字母分别表示色调、饱和度和亮度Brightness)。HSV模型的优点是,像颜色“调亮”或“调暗”这样的概念可以清晰地反映到这种模型上——比如,可以只调整饱和度(如图3.5所示)。另一种模型是CMYK颜色模型,它编码了青色(Cyan)、紫红色(Magenta)、黄色(Yellow)和黑色(blacK)(使用字母K而不是B是因为B会与“蓝色”(Blue)混淆)。CMYK是打印机使用的模型——4种颜色对应打印机混合产生颜色的4种墨水。然而,使用4种元素意味着需要在计算机上编码更多内容,因此,这种模型在媒体计算中不那么流行。RGB是计算机上最流行的模型。

像素的每种颜色分量(有时也称为颜色通道)通常都以单个字节表示,一个字节包含8位。8位可以表示256种模式(2^8):从00000000, 00000001, ..., 11111111。通常,我们使用这些模式来表示数值0~255。于是,每个像素就使用24位表示颜色。24位总共可以有 2^{24} 种可能的0/1组合模式,这意味着使用RGB模型的标准颜色编码可以表示16 777 216种颜色。实际上,我们确实能感知超过1 600万种颜色,但这并不重要。要完全复制我们能够看到的颜色空间,还没有哪种技术能接近这一目标。我们确实有能够表示1 600万种不同颜色的设备,但这1 600万种颜色也不能涵盖所有我们能够感知的颜色(或亮度)空间。因此,在技术取得进展之前,24位的RGB模型就足够了。

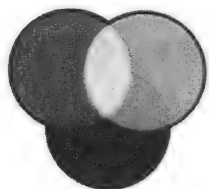


图3.4 融合红、绿、蓝产生新颜色

在有些计算机模型中,每个像素会使用更多的位。比如,有一种32位的模型使用另外的8位来表示透明度(transparency),即给定图像“底下”的颜色应该有多少与图像本身的颜色混合起来。这额外的8位有时称为alpha通道(alpha channel)。也有红、绿、蓝三个通道各使用超过8位的模型,但并不常见。

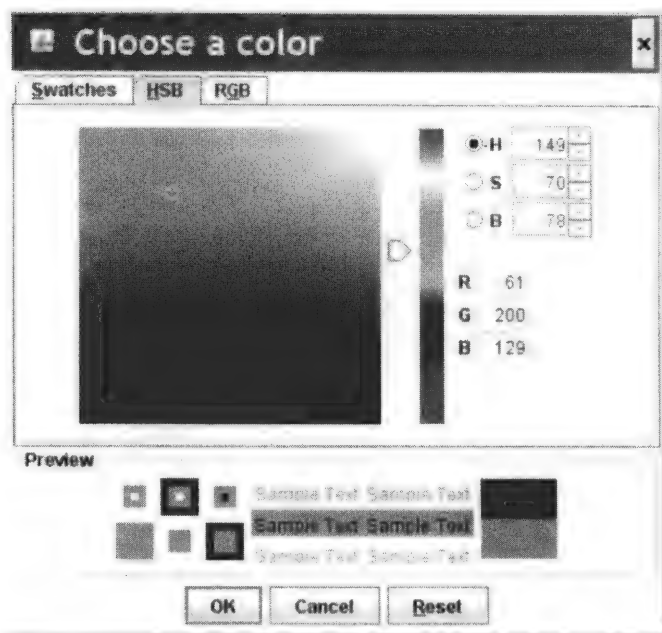


图3.5 基于HSB颜色模型选取颜色

实际上，我们有单独的一套视觉系统来感知物体的边缘、运动和厚度。我们通过一套系统来感知色彩，通过另一套系统来感知亮度（luminance，东西有多亮或多暗）。亮度实际上不是光量（amount of light），而是我们对光量的感知。我们可以度量光量（比如，基于颜色反射的光子数目）并验证一个红点与一个蓝点反射的光量是一样的，但我们还是感觉蓝色更暗。我们的亮度感基于物体与周围环境的对比。图3.6中的视觉错觉突出展现了我们是如何感知灰度级别的。左右两端实际上是同一灰度级，但因为中间的部分亮度和暗亮反差很大，所以我们感觉一端比另一端更暗。

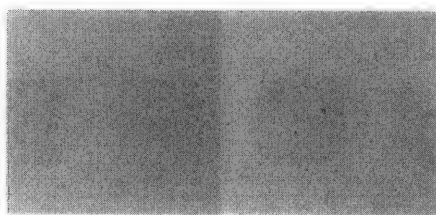


图3.6 此图两端是同样的灰色，但中间的部分对比明显，因此左端看上去比右端更暗

大多数让用户选取颜色的工具都允许用户以RGB分量的形式指定颜色。JES中的颜色选取器（实际是Java Swing的标准颜色选取器）提供了一组滑块，可用于控制每种颜色的分量（如图3.7所示）。可以在命令区输入以下命令来选取颜色。

```
>>> pickAColor()
```

前面提过，三元组（0，0，0）（分别对应红、绿、蓝分量）是黑色，而（255，255，255）是白色。（255，0，0）是红色，（100，0，0）也是红色——只是更暗一些。（0，100，0）是中等亮度的绿色，而（0，0，100）是中等亮度的蓝色。当红、绿、蓝三种分量相同的时候，结果就是灰色。（50，50，50）是相当暗的灰色，而（100，100，100）更亮一些。

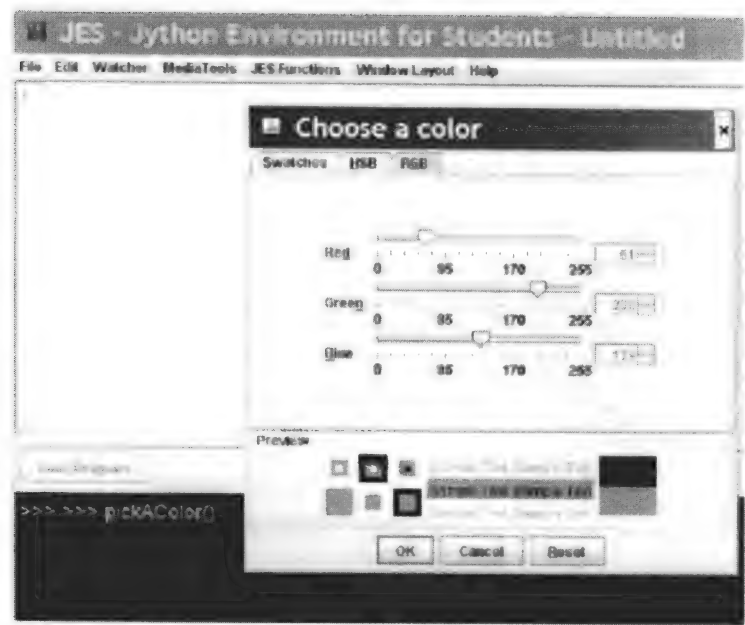


图3.7 JES中使用RGB滑块选取颜色

图3.8用一个矩阵表示像素RGB三元组。在图3.8中，(1, 0)处的像素颜色是(30, 30, 255)，意味着它有红色值30，绿色值30和蓝色值255——基本上它是一种蓝色，只是不纯。(2, 1)处的像素颜色是(150, 255, 150)，它有纯绿的颜色值，但也有更多的红和蓝，因此是一种很淡的绿色。

磁盘上甚至计算机内存中的图像通常都以压缩形式存储。即使一幅小图片，要表示它的每个像素也需要数量可观的内存（如表3.1所示）。一幅很小的320×240像素，每像素24位的图片占用的内存也有230 400字节——大约230 KB或1/4 MB。一台宽1024像素、高768像素的计算机显示器，每像素32位，要表示整屏画面需要3 MB。

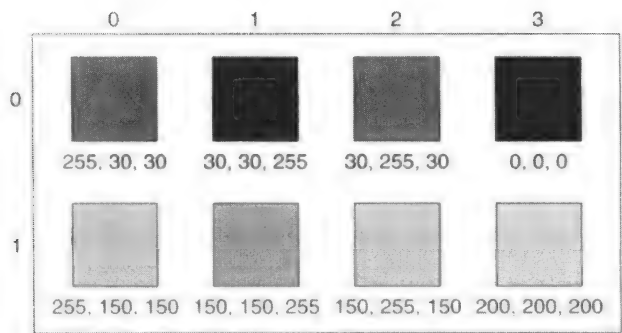


图3.8 以矩阵表示的RGB三元组

表3.1 存储不同尺寸和格式的图片像素时所需的字节数

	320×240图像	640×480图像	1024×768图像
24位彩色	230 400字节	921 600字节	2 359 296字节
32位彩色	307 200字节	1 228 800字节	3 145 728字节

3.2 处理图片

在JES中，我们从JPEG文件中构造出图片对象，然后改变图片中各像素的颜色，以此来处理图片。为改变像素的颜色，我们需要处理像素的红、绿、蓝分量。

我们用makePicture函数来构造图片，使用show来显示图片。

```
>>> file = pickAFile()
>>> print file
C:\ip-book\mediasources\beach.jpg
>>> myPict = makePicture(file)
>>> show(myPict)
>>> print myPict
Picture, filename C:\ip-book\mediasources\beach.jpg
      height 480 width 640
```

makePicture函数完成的工作是：接收输入文件名中的所有字节，将它们放入内存，稍微再次调整格式，然后为它们打上一个标记，宣称“这是一幅图片！”执行myPict = makePicture(filename)命令就等于说：“那个图片对象（注意它上面的标记）现在叫做myPict”。

图片知道自己的宽度和高度，可以用getWidth和getHeight来查询。

```
>>> print getWidth(myPict)
640
>>> print getHeight(myPict)
480
```

有了图片对象，再给出目标像素的坐标，我们就可以用getPixel函数来获取图片中的任何一个像素。我们还可以用getPixels来获得包含所有像素的一维数组。一维数组从第一行的所有像素开始，后面跟着第二行的所有像素，以此类推。我们使用“[下标]”的形式引用数组元素。

```
>>> pixel = getPixel(myPict,0,0)
>>> print pixel
Pixel red=2 green=4 blue=3
>>> pixels = getPixels(myPict)
>>> print pixels[0]
Pixel red=2 green=4 blue=3
```



常见bug：不要输出像素数组，它太大了

getPixels毫无保留地返回了全部像素组成的数组。如果你试图输出getPixels函数的返回值，就会输出所有的像素。图片中的像素有多少呢？好吧，告诉你“beach.jpg”的宽640、高480。那会输出多少行呢？ $640 \times 480 = 307\,200$ ！307 200行的输出非常庞大。很可能你根本没有耐心等待它结束。如果不小心做了这种事情，直接退出JES重启吧。

像素知道自己来自何处。可以用getX和getY询问它们的x和y坐标。

```
>>> print getX(pixel)
0
>>> print getY(pixel)
0
```


各个像素还知道如何getRed、setRed。(绿和蓝类似。)

```
>>> print getRed(pixel)
2
>>> setRed(pixel,255)
>>> print getRed(pixel)
255
```

也可以用getColor询问像素的颜色。还可以用setColor设置它的颜色。颜色对象知道自己的红、绿、蓝颜色分量。可以用函数makeColor来构造新的颜色。

```
>>> color = getColor(pixel)
>>> print color
color r=255 g=4 b=3
>>> newColor = makeColor(0,100,0)
>>> print newColor
color r=0 g=100 b=0
>>> setColor(pixel,newColor)
>>> print getColor(pixel)
color r=0 g=100 b=0
```

如果像素的颜色改变了，像素所属的图片也会改变。

```
>>> print getPixel(mypicture,0,0)
Pixel, color=color r=0 g=100 b=0
```

常见bug：观看图片的变化



如果你显示了图片，然后改变了像素，你可能会疑惑为什么没看出任何不同。图片显示是不会自动更新的。对图片执行repaint，比如repaint(picture)，它才会更新。

也可以用pickAColor构造颜色，这个函数提供了多种选取颜色的方法。

```
>>> color2=pickAColor()
>>> print color2
color r=255 g=51 b=51
```

处理完一幅图片之后，可以用writePictureTo函数把它写到文件中。

```
>>> writePictureTo(myPict,"C:/temp/changedPict.jpg")
```



常见bug：以“.jpg”结尾

一定要用“.jpg”作为文件名的结尾，以便操作系统把它识别为JPEG文件。



常见bug：快速保存文件——以及怎样再次找到它

如果不知道所选目录的完整路径该怎么办呢？你可以仅指定文件基本名。

```
>>> writePictureTo(myPict,"new-picture.jpg")
```

问题在于如何再次找到这个文件。它保存到哪个目录里去了？这是个很容易解决的bug。默认目录（未指定路径时使用的目录）是JES所在的目录。如果不久前使用过pickAFile()，默认目录就是从中选取文件的目录。如果你有一个保存媒体并从中选取文件的标准媒体文件夹

(比如, Mediasources), 那么, 当未指定完整路径时, 它就是保存文件的目录。

我们无须编写新的函数来处理图片, 可以在命令区直接使用前面描述的函数。

```
>>> file="C:/ip-book/mediasources/caterpillar.jpg"
>>> pict=makePicture(file)
>>> show(pict)
>>> pixels = getPixels(pict)
>>> setColor(pixels[0],black)
>>> setColor(pixels[1],black)
>>> setColor(pixels[2],black)
>>> setColor(pixels[3],black)
>>> setColor(pixels[4],black)
>>> setColor(pixels[5],black)
>>> setColor(pixels[6],black)
>>> setColor(pixels[7],black)
>>> setColor(pixels[8],black)
>>> setColor(pixels[9],black)
>>> repaint(pict)
```

结果如图3.9所示, 图片的左上方显示了一条短短的黑线, 这条黑线的长度为10个像素。

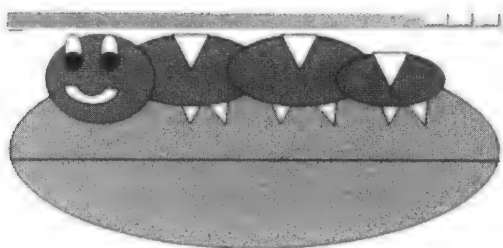
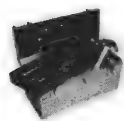


图3.9 使用命令直接修改像素颜色: 注意左上角的短小黑线



实践技巧: 使用JES帮助

JES拥有极好的帮助系统。忘记了自己需要哪个函数? 直接从JES的Help菜单上选择一项就行了(图3.10——全是超链接, 需要什么自己搜寻一下吧)。忘记了自己用过的某个函数是干什么的? 选中它, 然后选择Help菜单中的Explain就可以得到针对选中内容的解释(如图3.11所示)。

浏览图片

可以到<http://mediacomputation.org>上找到MediaTools应用程序以及关于如何使用它的文档。还可以在JES中找到MediaTools菜单。两种MediaTools都有一组图片浏览工具, 对研究图片非常有用。MediaTools应用程序如图3.12所示。JES图片工具如图3.13所示。

JES图片工具基于在命令区定义并命名的图片对象来工作。如果没有给图片命名, 就不能用JES图片工具来查看它。`p = makePicture(pickAFile())`定义一个图片对象并将它命名为p。然后你就能浏览图片了, 可以用`explore(p)`, 也可以用MediaTools菜单中的Picture Tool, 浏览的时候会有一个菜单弹出, 上面有当前可用的图片对象(基于其变量名), 可以选择其中的一个并点击OK按钮(如图3.14所示)。

JES图片工具允许你浏览图片。可以从Zoom菜单中选择一个级别来放大或缩小图片。在鼠标移过图片时按下鼠标按钮可以显示当前所指像素的(x, y)(横, 纵)坐标和RGB值(如

图3.13所示)。

Table of Contents > Understanding Pictures in JES > Picture Objects in JES	
Picture Objects in JES	
To understand how to work with pictures in JES, you must first understand the objects (or encodings) that represent pictures.	
You can imagine that each picture is made up of a collection of pixels , which is made up of pixel 1 , pixel 2 , pixel 3 , etc, and that each pixel has it's own particular color .	
Pictures	Pictures are encodings of images, typically coming from a JPEG file.
Pixels	Pixels are a sequence of Pixel objects. They flatten the two dimensional nature of the pixels in a picture and give you instead an arraylike sequence of pixels. pixels[1] returns the leftmost pixel in a picture.
Pixel	A pixel is a dot in the Picture. It has a color and an (x, y) position associated with it. It remembers its own Picture so that a change to the pixel changes the real dot in the picture.
Color	It's a mixture of red, green, and blue values, each between 0 and 255.
▲ jump to top	

图3.10 JES帮助条目示例

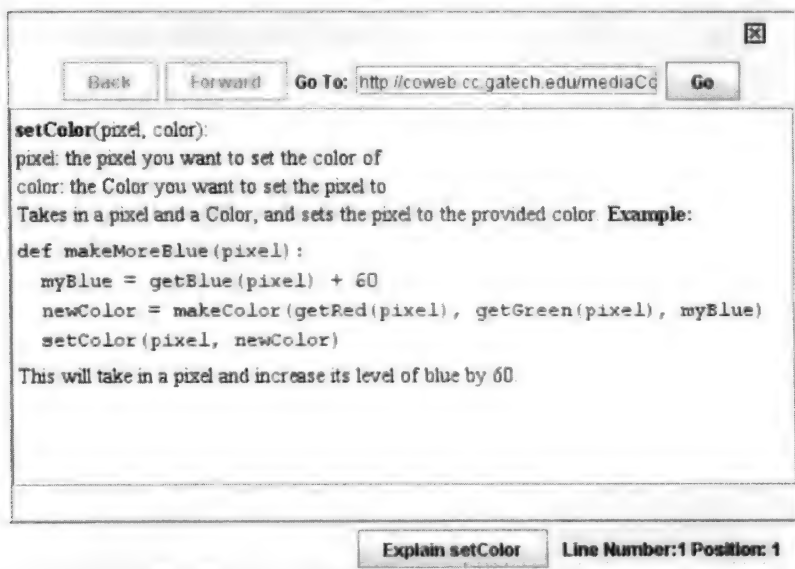


图3.11 JES Explain条目示例



图3.12 使用MediaTools图像浏览工具

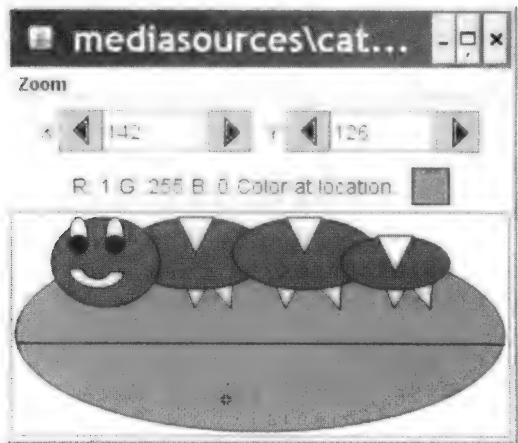


图3.13 在JES图片工具中选择像素

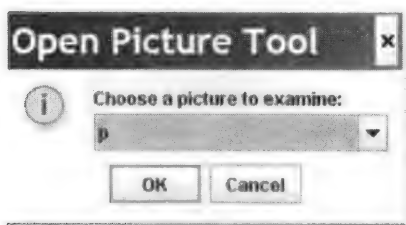


图3.14 在JES图片工具中选择图片

MediaTools应用程序基于磁盘文件来工作，而不是基于命令区中命名的图片来工作。如果你想先在文件加载到JES之前先检查一下，那么可以用MediaTools应用程序做这件事。点击MediaTools中的Picture Tools就可以打开图片工具，可以用Open按钮打开一个文件选择框——

从左边点击想要浏览的目录，从右边点击想要浏览的文件，然后点击OK按钮。图像显示出来以后，还有多种不同的工具可供选用。把鼠标移到图片上并用鼠标按钮按下：

- 鼠标当前所指像素的红、绿、蓝颜色值将显示出来。如果你想对图片如何映射成红、绿、蓝颜色数值有一些感性认识，那么这一功能非常有用。如果你想对像素做些计算并检查结果值，这也很有帮助。
- 鼠标当前所指像素的 x 和 y 坐标将显示出来。当你想知道屏幕上一块区域的坐标范围时（比如，你只想处理图片的一部分）它很有用。如果你想处理某块区域而且知道这块区域的 x 和 y 坐标范围，那么就可以适当地安排一个for循环，只处理这一部分。
- 最后，还会显示出一个放大镜，让你看到像素逐渐放大的效果。（放大镜可以点击、拖曳。）

3.3 改变颜色值

最简单的图片处理是通过改变像素的红、绿、蓝分量来改变图片中像素的颜色值。只需对这些值做简单调整，就能得到迥然不同的图片效果。Adobe Photoshop的滤镜（filter）所做的工作就是我们在这一节要做的事情。

我们将要使用的颜色处理方法是基于原始颜色计算一个百分比。如果想得到原始图片中50%的红色数量，我们会把红色通道设置为当前值的0.5倍。如果想把红色增加25%，则可以把红色设为当前值的1.25倍。别忘记Python中的乘法运算符是星号（*）。

3.3.1 在图片上运用循环

我们可以获得图片中的各个像素并将它们的红、绿或蓝分量设置为新值。比如，我们想把红色减少50%，肯定可以这样编写代码：

```
>>> file="C:/ip-book/mediasources/barbara.jpg"
>>> pict=makePicture(file)
>>> show(pict)
>>> pixels = getPixels(pict)
>>> setRed(pixels[0],getRed(pixels[0]) * 0.5)
>>> setRed(pixels[1],getRed(pixels[1]) * 0.5)
>>> setRed(pixels[2],getRed(pixels[2]) * 0.5)
>>> setRed(pixels[3],getRed(pixels[3]) * 0.5)
>>> setRed(pixels[4],getRed(pixels[4]) * 0.5)
>>> setRed(pixels[5],getRed(pixels[5]) * 0.5)
>>> repaint(pict)
```

但这么写太枯燥了，特别当针对所有像素的时候，哪怕对小图片也是一样。我们需要的是让计算机反反复复做同样一件事情的方法。当然，并非完全同样的事情——我们想以一种明确定义的方法来调整正在进行的过程。每次只运行一步，或者说只处理一个像素。

我们可以用for循环来达到这一目的。for循环针对（你提供的）数组中的每个项执行（你指定的）某个命令块，每次执行命令时，一个（你命名的）特别的变量将持有数组中不同元素的值。数组是数据的顺序集合。getPixels返回的是包含输入图片中所有像素对象的数组。

我们将写出下面这样的语句：

```
for pixel in getPixels(picture):
```

我们来详细讨论一下这里的每一部分。

- 首先是命令的名字for。
- 接下来是你想在寻址 (addressing) (并处理) 序列元素的代码中使用的变量名字。我们在这里使用单词pixel, 因为我们想处理图片中的每个像素。
- 单词in是需要的——你一定要输入这个单词! 输入它, 命令的可读性更好, 所以, 多敲这4下键盘 (空格-i-n-空格) 是有益的。
- 接下来需要一个数组。在每一轮循环中, 变量pixel将赋给数组中的每个元素: 一个元素循环迭代一次。使用getPixels函数来产生这个数组。
- 最后, 需要有个冒号 (“:”)。这个冒号很重要——它表示后续内容是一个块 (block) (你应该还记得上一章介绍的有关“块”的知识)。

接下来便是你想针对各个像素执行的命令。每次执行命令时, 变量 (在这个例子中是pixel) 会引用数组中的不同元素。这些命令 (称为循环体) 作为一个块指定。这意味着它们应该跟在for语句之后, 每条占一行, 而且要比for语句多缩进两个空格! 例如, 下面就是把各像素的红色通道设置为原值一半的for循环。

```
for pixel in getPixels(picture):
    value = getRed(pixel)
    setRed(pixel, value * 0.5)
```

我们把这段代码查走一遍。

- 第一条语句表明我们将拥有一个for循环, 它会把变量pixel设置为getPixels(picture)所输出数组的每个元素。
- 下一条语句向右缩进, 因此它是for语句循环体的一部分——每次变量pixel拥有新值时将执行的语句之一 (不论图片中的下一个像素是什么)。它取得当前像素的当前红色值并把它保存在变量value中。
- 第三条语句仍然向右缩进, 因此它仍是循环体的一部分。在这一行, 把名为pixel的像素的红色通道值设置为变量value乘以0.5的结果 (使用setRed()函数)。这将使原值减少一半。

别忘了用来定义函数的def语句之后也是一个块。如果某函数中有一个for循环, 那么for语句已经缩进两个空格了, 因此for的循环体 (循环中执行的命令) 必须缩进4个空格。for循环的块位于函数块的内部。这叫做内嵌块 (nested block) ——一个块嵌套在另一个块中。下面的例子把循环语句变成了一个函数。

```
def decreaseRed(picture):
    for pixel in getPixels(picture):
        value = getRed(pixel)
        setRed(pixel, value * 0.5)
```

实际上, 不需要把循环放到函数中使用。可以在JES的命令区直接输入它们。JES很智能, 如果输入的是循环命令, 它能猜出需要输入多条命令, 因此会将提示符 “>>>” 变为 “...”。当然, 它无法猜出什么时候结束, 因此, 需要直接按Enter键 (不输入其他任何东西), 以此告诉JES循环体已经结束。你可能意识到了, 其实不需要变量value——可以用能求出相同值的函数调用来替代这个变量。在命令区这样输入:

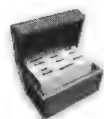

```
>>> for pixel in getPixels(picture):
...     setRed(pixel, getRed(pixel) * 0.5)
```

我们已经明白了如何在不用编写上千行代码的情况下让计算机执行上千条命令，那就试试吧。

3.3.2 增/减红（绿、蓝）

处理数字图片的一种常见需求就是改变图片的红色度（或者绿色度、蓝色度——但更常见的是红色度）。可以把红色度提高，从而“暖化”图片，也可以降低它来“冷却”图片，或者应对红色过度的数码相机。

下面的菜谱将输入图片的红色数量减少50%。它用变量p代表当前像素。我们在上一个函数中使用了变量pixel。这无关紧要——名字可以随意取。



程序9：将图片中的红色数量减少50%

```
def decreaseRed(picture):
    for p in getPixels(picture):
        value=getRed(p)
        setRed(p,value*0.5)
```

把上面这段程序直接输入JES的程序区。点击Load Program按钮，以便让Python处理这个函数（一定要保存它，比如保存为DECREASERED.PY，或类似的名字），从而使名字decreaseRed代表这个函数。可以一步步执行下面的示例，进一步弄清这一切的原理。

这份菜谱接受一幅图片作为输入——我们将从这幅图片中取得像素。为了得到图片，需要一个文件名，然后需要基于文件构造一幅图片。也可以使用函数explore(picture)基于图片对象打开JES图片工具。它会将当前图片复制一份并在JES图片工具中显示它。对图片应用decreaseRed函数之后，可以再次浏览它并让两幅图片并排在一起进行比较。因此，菜谱可以这样使用：

```
>>> file="C:/ip-book/mediasources/Katie-smaller.jpg"
>>> picture=makePicture(file)
>>> explore(picture)
>>> decreaseRed(picture)
>>> explore(picture)
```



常见bug：耐心一点——for循环总会结束的

对这种代码来说，最常见的bug就是不等它自己结束就按下了Stop按钮。如果你使用的是for循环，程序总会结束的。但对于我们执行的一些操作，它可能需要运行整整一分钟（或者两分钟！）——特别是源图像很大的时候。

原始图片及其减少红色的版本如图3.15所示。50%显然是个不小的比例。图片看上去像是经过蓝色滤镜拍下的。注意第一个像素的红色量原来是133，后来变成了66。

跟踪程序：程序是如何执行的

计算机科学思想：跟踪是最重要的技能

编程实践中你能练就的最重要的一项技能就是跟踪程序（有时也称为程序的单



步跟踪)。跟踪程序就是一行一行地执行它，弄清楚每一行发生了什么。看一个程序时，你能预知它的功能吗？你应该能够逐行想出它的功能。

程序原理

让我们跟踪一下这个减少红色的函数，看看它的工作原理。我们想从刚刚调用 `decreaseRed` 的地方打断并开始跟踪。

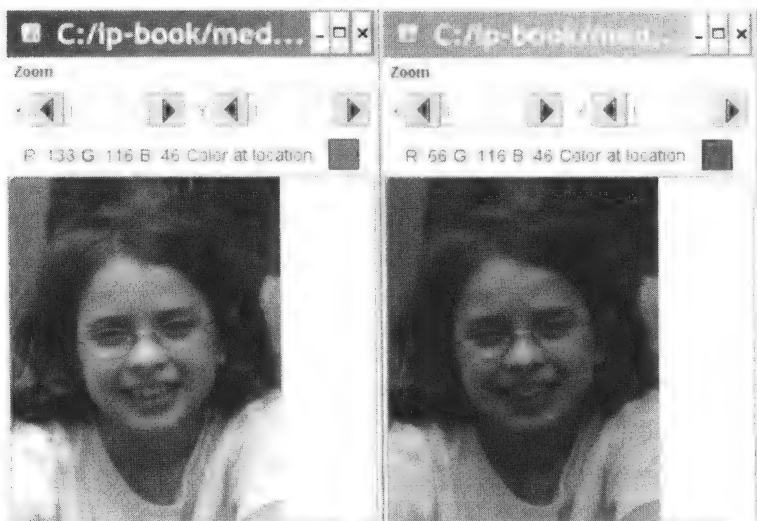


图3.15 原来的图片（左）和减少红色的版本（右）

```
>>> file="C:/ip-book/mediasources/Katie-smaller.jpg"
>>> picture=makePicture(file)
>>> explore(picture)
>>> decreaseRed(picture)
>>> explore(picture)
```

这时发生了什么？名字 `decreaseRed` 的确代表了之前看到的函数，于是它开始执行。

```
def decreaseRed(picture):
    for p in getPixels(picture):
        value=getRed(p)
        setRed(p,value*0.5)
```

第一行执行的是 `"def decreaseRed(picture):"`。这一行指明了函数接受某种输入，而且在函数执行过程中输入将命名为 `picture`。

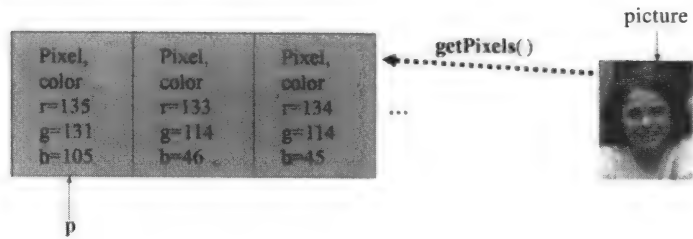
计算机科学思想：函数内部的名字与函数外面的名字是不一样的

函数内部的名字（如 `decreaseRed` 例子中的 `picture`、`p` 和 `value`）与命令区中的名字以及任何其他函数中的名字是完全不同的。我们说：它们有不同的作用域（scope）。

可以想象，这个时候计算机内部的样子：在单词 `picture` 和我们输入的图片对象之间存在某种关联。

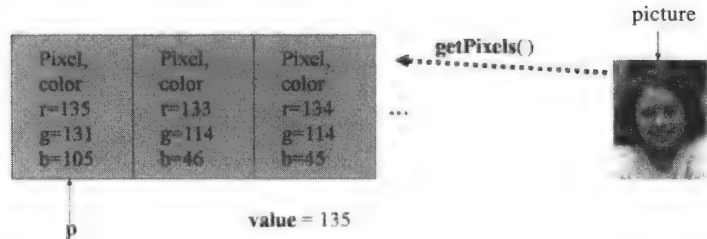


现在我们执行到了下一行：“for p in getPixels(*picture*):”。这一行的意思是：来自图片的所有像素都（在计算机内部）排成一个序列（位于数组中），且变量*p*应该赋予（关联到）第一个元素。可以想象，此时计算机内部就像这个样子：

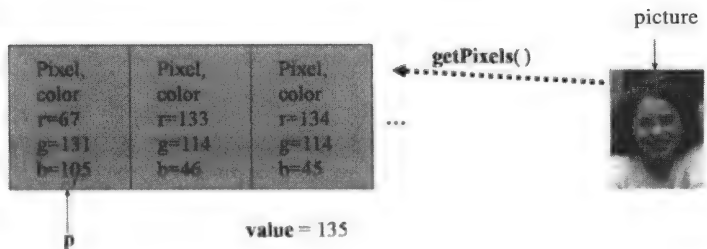


每个像素都有自己的RGB值。*p*指向第一个元素。注意变量*picture*还在那儿——我们用它或者不用它，它都在那里。

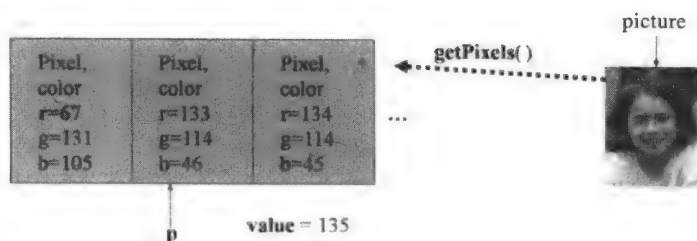
现在到了 `value = getRed(p)` 这一行。它只是把另一个名字加进了计算机正为我们跟踪的名字集合中，并为这个名字提供了一个简单的数字值。



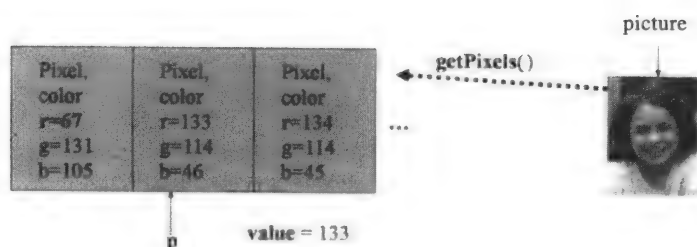
最后，我们到了循环的末尾。计算机执行 `setRed(p, value * 0.5)`，把像素*p*的红色通道改为*value*的50%。*p*的值是奇数，乘以0.5会得到67.5，但我们把结果放进一个整数中，因此小数部分就被丢弃了。原先的红色值135变成了67。



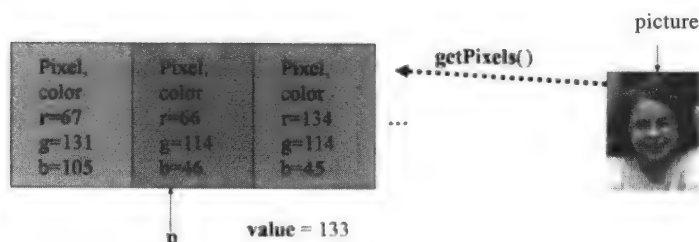
接下来发生的事情非常重要：循环又重新开始了！我们又回到了for循环并取出了数组中的下一个值。名字*p*与下一个值关联了起来。



在`value = getRed(p)`这一步，变量`value`又获得了新值，现在它是133，而不是上次来自第一个像素的135。



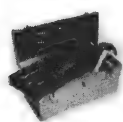
然后，我们又改变了这个像素的红色通道。



最终，我们得到了图3.15。我们遍历了序列中的所有像素并修改了全部红色值。

3.3.3 测试程序：它真的能运行吗

我们如何知道自己所做的一切真正有效呢？当然，图片发生了一些变化，但我们真的把红色减少了吗？而且是50%？



实践技巧：不要轻易相信自己的程序

人们很容易误认为自己的程序已经正常工作了。毕竟，你让计算机做一些事，如果它做了你让它做的，那么你就不会觉得奇怪。但计算机真的很愚蠢——它们猜不出你想要什么。它们只是做你让它们做的事。你很容易得到“几乎正确”的结果。一定要检查一下。

检查程序的方法可以有多种。一种方法是使用JES的图片工具。用图片工具可以检查之前和之后的图片在同一 x 和 y 坐标处的RGB值。在之前的图片中点击并拖动光标到某个点来检查，然后在之后的图片中输入相同的 x 和 y 坐标并按Enter键。这样你可以同时看到之前和之后的图片在 x 和 y 坐标处的颜色值（如图3.16所示）。

我们也可以在命令区使用之前已知的函数来检查各个像素的红色数量。

```

>>> file = pickAFile()
>>> pict = makePicture(file)
>>> pixel = getPixel(pict,0,0)
>>> print pixel
Pixel, color=color r=168 g=131 b=105
>>> decreaseRed(pict)
>>> newPixel = getPixel(pict,0,0)
>>> print newPixel
Pixel, color=color r=84 g=131 b=105
>>> print 168 * 0.5
84.0

```

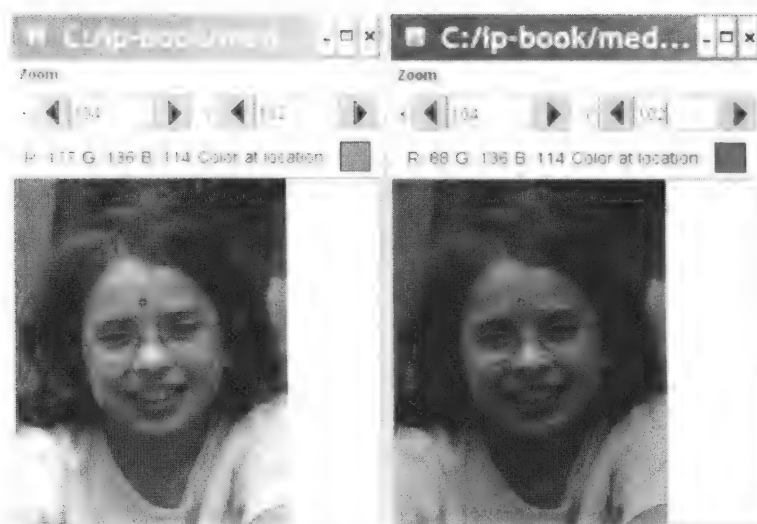


图3.16 使用JES图片工具让自己确信红色已经减少

3.3.4 一次修改一种颜色

现在让我们增加图片中的红色。如果将红色分量乘以0.5能使之减少，那么乘以一个大于1.0的数就能使之增加。



程序10：将红色分量增加20%

```

def increaseRed(picture):
    for p in getPixels(picture):
        value=getRed(p)
        setRed(p,value*1.2)

```

程序原理

就与使用decreaseRed一样，我们将使用同样的命令区（Command Area）语句来使用increaseRed。当我们输入increaseRed(picture)这样的命令时，会发生同样的过程。我们获得了输入图片picture的所有像素（不论它是什么样的图片），然后将变量p赋予列表中的第一个像素。我们取得它的红色值（假如说是100）并命名为value。我们将名字p当前所代表像素的红色值赋为 $1.2 * 100$ ，或者说120。然后我们针对输入图片中的每一个像素p重复这一过程。

如果一幅图片中的红色数量很多，增加它会导致某些红色结果值超出255，那么这时的情

况会怎样呢？针对这种情况有两种选择。结果可以截断为最大值255，也可以使用模（余数）运算符来回绕（wrap around）。例如，如果当前值为200而你试图将它倍增至400，那么它可以保持为255，或者绕回为144 ($400 - 256$)。尝试一下，看看会有什么效果。

JES提供了一个选项来控制截断颜色值为255还是回绕它。要改变此选项可以点击Edit菜单，然后点击Options。需要改变的选项是：Mouulo pixel color values by 256。

我们甚至可以完全消除一种颜色分量。下面的菜谱清除了一幅图片的蓝色分量。



程序11：清除图片的蓝色分量

```
def clearBlue(picture):
    for p in getPixels(picture):
        setBlue(p,0)
```

3.4 制作日落效果

我们当然可以同时完成多种图片处理动作。有一次，Mark想基于一幅海滩风景产生一种日落效果。他的第一次尝试是增加红色量，但没有达到效果。在一幅给定的图片中，某些像素的红色量已经很高了。如果某个通道的值超出255，那么默认结果是回绕。比如，通过setRed()函数把某个像素设置成256，实际会得到0。因此，增加红色会产生明亮的蓝、绿（没有红色）点。

Mark的第二种想法是：或许日落中的效果是蓝色和绿色更少，从而突出了红色，而不是实际增加了红色。下面是他针对这种想法编写的程序：



程序12：制作日落效果

```
def makeSunset(picture):
    for p in getPixels(picture):
        value=getBlue(p)
        setBlue(p,value*0.7)
        value=getGreen(p)
        setGreen(p,value*0.7)
```

程序原理

与前面的例子一样，我们接受picture输入并用变量p表示输入图片中的各个像素。我们取得各个像素的蓝色分量并重设其值为原值乘以0.70的积。然后对绿色做同样的操作。结果，我们同时修改了绿色和蓝色通道——分别减少30%。效果非常不错，如图3.17所示。

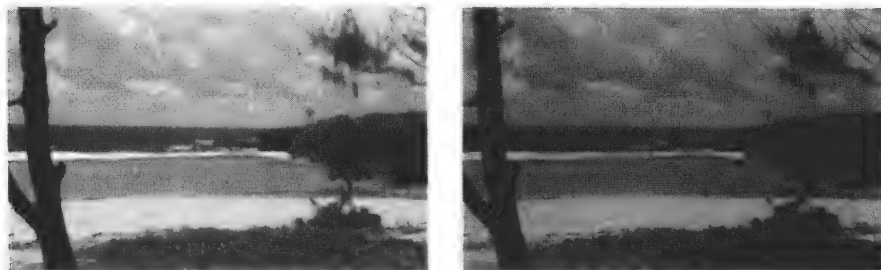


图3.17 原来的海滩风景（左）和（虚假的）日落时的效果（右）

理解函数

此时此刻，关于函数你可能有许多问题要问。我们为什么以这种方式编写这些函数？我们为什么能同时在函数和命令区中重用`picture`这样的变量名？编写这些函数还有其他方法吗？函数有好坏之分吗？

由于我们总是先选取一个文件（或输入一个文件名），然后在调用某个文件处理函数之前先构造一幅图片，然后再显示或浏览图片，你自然会问：为什么不把这些固定下来？为什么不让每个函数直接包含`pickAFile()`和`makePicture()`？

我们使用函数的方式是使之更加通用或可重用。我们想让每个函数做一件事且只做一件事，于是我们可以在另一个需要做这件事情的上下文中再次使用这个函数。举个例子会更清楚。考虑一下制作日落效果的程序（程序12）。它通过将绿色和蓝色分别减少30%来达到效果。如果我们重写这个函数，让它调用两个更小的、分别只完成这两种操作的函数，那么结果会怎样呢？我们可能编写像程序13这样的代码。



程序13：使用三个函数制作日落效果

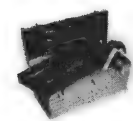
```
def makeSunset2(picture):
    reduceBlue(picture)
    reduceGreen(picture)

def reduceBlue(picture):
    for p in getPixels(picture):
        value=getBlue(p)
        setBlue(p,value*0.7)

def reduceGreen(picture):
    for p in getPixels(picture):
        value=getGreen(p)
        setGreen(p,value*0.7)
```

程序原理

首先要弄清楚的是：这确实能达到了效果。这里的`makeSunset2`做的事情与之前的菜谱一样。`makeSunset2`函数接受一幅输入图片，然后使用同一幅输入图片调用了`reduceBlue`。`reduceBlue`使用`p`表示输入图片中的各个像素，并将每个像素的蓝色减少了30%（乘以0.7）。然后`reduceBlue`结束了，程序的控制流（即，接下来要执行的语句）返回`makeSunset2`函数并执行下一条语句。下一条语句是用同一幅输入图片调用函数`reduceGreen`。与前面一样，`reduceGreen`遍历每个像素并将绿色值减少30%。



实践技巧：使用多个函数

在同一程序区中使用多个函数并将它们保存在同一文件中完全没有问题。这将使函数的阅读和重用更加容易。

让一个函数（本例中是`makeSunset2`）使用程序员在同一文件中编写的其他函数（`reduceBlue`和`reduceGreen`）完全没有问题。你可以与之前一样使用`makeSunset2`。它还是同一份菜谱（让计算机做同样的事情），但使用了不同的函数。实际上，你也可以直接使用`reduceBlue`和`reduceGreen`——在命令区构造一幅图片并作为输入传给两个函数。它们用起来就与`decreaseRed`一样。

不同的是，makeSunset2的可读性更好一些，它清晰地表达了“制作日落效果就是减少蓝绿两种颜色”的意思。方便阅读真的很重要。

计算机科学思想：程序是以人为本的

计算机对程序的外表根本不关心。程序写出来是为了与人交流。程序易阅读易理解意味着它们易修改易重用，而且能更有效地将过程传达给其他人。

如果我们在reduceBlue和reduceGreen中加入pickAFile、show和repaint又会怎样？那样我们需要将图片提供两次——每个函数一次。把函数写成只减少蓝色或减少绿色（做且只做一件事），我们就可以在makeSunset2这样的新函数中使用它们。

现在，假如我们把pickAFile和makePicture放进makeSunset中。reduceBlue和reduceGreen函数再次变得完全灵活且可重用了，这很重要吗？没有，如果你只关心一个能给单幅图片带来日落效果的函数，那它就不重要。但如果以后你想制作一段几百帧的电影，为每一帧加入日落效果呢？你会愿意把那几百帧图像中的每一帧都取出来处理一遍？还是编写一个循环遍历它们（后面章节会学到），并将每一帧都作为输入传给更通用的makeSunset更好呢？这就是为什么我们要把函数做得通用且可重用——你永远不知道自己什么时候会在更大的上下文中再次使用一个函数。

实践技巧：开始时不要急着编写应用程序

新程序员常常想编写可供非技术用户使用的完整应用程序。你可能考虑编写一个makeSunset应用程序，运行时帮用户取得一幅图片并生成日落效果。构建大家都可以使用的一套良好用户界面并非易事。刚开始还是慢慢来比较好。编写一个接收图片输入的可重用函数已经够难的了。你可以将来再考虑用户界面。

也可以使用显式的文件名来编写这些函数，方法是在程序开头写：

```
file="C:/ip-book/mediasources/bridge.jpg"
```

这样，程序就不会每次都提示我们提供一个文件，但函数也只能处理一个文件了。如果想让它处理其他文件，那么我们必须修改程序。你会愿意每次使用一个函数时都改动它一下吗？还是让函数保持独立，每次只改动传给它的图片更方便。

当然，我们可以把任何一个函数都改成传入文件名而不是传入图片。比如，我们可以写：

```
def makeSunset3(filename):
    reduceBlue(filename)
    reduceGreen(filename)

def reduceBlue(filename):
    picture = makePicture(filename)
    for p in getPixels(picture):
        value=getBlue(p)
        setBlue(p,value*0.7)

def reduceGreen(filename):
    picture=makePicture(filename)
    for p in getPixels(picture):
        value=getGreen(p)
        setGreen(p,value*0.7)
```

比起之前的代码，这样做好还是不好呢？从某种层面来说，这无所谓——只要合理，我们可以使用图片，也可以使用文件名。不过以文件名作为输入的版本确实有几个缺点：首先，它不能达到目的！`picture`分别在`reduceGreen`和`reduceBlue`中构造出来，但后来没有保存，因此函数结束时它就丢失了。之前的`makeSunset2`版本（以及它调用的子函数）是通过副作用来生效的——函数没返回任何东西，但它直接改变了输入对象。

我们可以在每个函数结束时将文件保存到磁盘，从而修正图片丢失的问题，但这样一来函数就不是“做且只做一件事”了。另外，两次构造图片还有低效的问题。如果我们再添加保存动作，那就保存了图片两次，更低效了。还是那句话，最好的函数“做且只做一件事”。

像`makeSunset2`这样的大函数同样“做且只做一件事”。`makeSunset2`制作了一张看似日落时分的图片。它是通过减少绿色和蓝色来实现的。我们最终得到的是一个目标的层次结构——目标就是“做且只做一件事”。`makeSunset2`让其他两个函数各自完成一件事，从而完成了自己的一件事。我们把这种过程称为层次式分解（*hierarchical decomposition*）（将一个程序分解为更小的部分，然后继续分解更小的部分，直到得出一项容易实现的任务），这一机制非常强大，有了它你可以基于自己理解的部分来创建复杂的程序。

函数中的名字与命令区中的名字是完全隔离的。函数从命令区获得数据的唯一方法是把数据作为输入传递给函数。在函数内部，可以用任何想用的名字。首次在函数内部定义的名字（如上一例中的`picture`）和用来表示输入数据的名字（如`filename`）仅存在于函数执行的过程中。函数结束后这些变量名就不复存在了。

这实际是个优点。先前我们说过，命名对计算机科学家非常重要：从数据到函数，我们给各种各样的东西命名。但如果每个名字永远表示且仅表示一样东西，那么我们说不定会把名字用光。在自然语言中，单词在不同的上下文中可以表示不同的意思。（比如，“What do you mean?”是问“你要表达什么意思？”而“You are being mean!”的意思是“你太坏了！”）不同的函数就是不同的上下文——其中的名字可以与函数外的同一个名字含义不同。

有时你会把函数里面计算出来的结果返回给命令区或调用方。我们已经见过函数输出值的情况，比如`pickAFile`输出一个文件名。如果在函数内部执行`makePicture`，那么你可能考虑将函数中创建的图片输出到外面去。这可以使用`return`来实现，后面我们会详细讨论它。

为函数的输入取的名字可以理解成占位符（*placeholder*）。看到占位符的时候，把它想象成输入数据就可以了。因此，像下面这样的函数：

```
def decreaseRed(picture):
    for p in getPixels(picture):
        value=getRed(p)
        setRed(p,value*0.5)
```

调用的时候我们将使用`decreaseRed(myPicture)`这样的语句。不论`myPicture`里面是什么图片，在`decreaseRed`执行期间它的名字就是`picture`。在这几秒钟内，`decreaseRed`中的`picture`和命令区中的`myPicture`表示的是同一幅图片。改变一幅图片的像素也就改变了另一幅图片的像素。

我们刚刚讨论过了用不同方法实现功能相同的函数——有的方法好一点，有的方法差一点。编写函数还有其他方法，有些与前面的差不多，有些则好得多。让我们再来考察几种编写函数的方法。

我们可以一次传入多个输入。考虑下面这个`decreaseRed`版本：

```
def decreaseRed(picture, amount):
    for p in getPixels(picture):
        value=getRed(p)
        setRed(p,value*amount)
```

我们可以用`decreaseRed(myPicture, 0.25)`这样的命令来使用这个函数，它将把红色减少75%。我们可以用`decreaseRed(myPicture, 1.25)`把红色增加25%。或许，这个函数还是叫`changeRed`更好，因为它现在就是这样的——一种通用的改变图片中红色总量的方法。这是个很有用也很强大的函数。

还记得程序11中的这段代码吗？

```
def clearBlue(picture):
    for p in getPixels(picture):
        setBlue(p,0)
```

我们也可以用下面的方法写出同样的菜谱：

```
def clearBlue(picture):
    for p in getPixels(picture):
        value = getBlue(p)
        setBlue(p,value*0)
```

需要注意的是：这个函数与之前的菜谱做了完全一样的事情，它们都把所有像素的蓝色通道设置成0。后一个函数的优点之一在于它与我们看到的其他改变颜色的函数在形式上别无二致，从而更易于理解，这是有益的。效率上它略逊一筹——将蓝色值设置为0之前没必要先取得原来的值，要得到一个0也没必要把另一个值乘以0。这个函数的确做了一些没必要做的事情——它没有“做且只做一件事”。

3.5 亮化和暗化

将图片加亮或变暗非常容易。模式与之前的代码完全相同，但不是改变一种颜色分量，而是改变整体颜色。以下是图片亮化和暗化的菜谱。图3.18显示了原始图片和更暗的版本。



程序14：亮化图片

```
def lighten(picture):
    for px in getPixels(picture):
        color = getColor(px)
        color = makeLighter(color)
        setColor(px,color)
```

程序原理

变量`px`用于表示输入图片中的各个像素。（这次不是`p`！这重要吗？计算机无所谓——如果你觉得`p`表示像素，那就用`p`，但`px`、`px1`甚至`pixel`也都可以，随便使用。）`color`接受了像素`px`的颜色值。`makeLighter`返回更亮的新颜色。`setColor`方法（面向对象术语中，与类或对象关联的函数常称为“成员函数”，也叫“方法”。作者这里冷不丁冒出“方法”一词，可能有些突兀。第16章将介绍面向对象编程。——译者注）将像素设成了更亮的新颜色。



图3.18 原始图片（左）和更暗的版本（右）



程序15：暗化图片

```
def darken(picture):  
    for px in getPixels(picture):  
        color = getColor(px)  
        color = makeDarker(color)  
        setColor(px,color)
```

3.6 制作底片

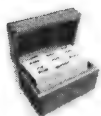
为图片制作一张底片像（negative image）比你想象的要容易得多。我们先来仔细考虑一下。想要达到的目标是把各像素的红、绿、蓝当前值都反一下。最容易理解的是边缘情形。倘若某个红色分量为0，我们想得到的是255。如果为255，我们则想把它变成0。

现在考虑中间地带。如果红色分量为淡红（比如50），我们想要的是一种近乎完全的红色——而“近乎”的程度应等同于原始图片中的红色数量。我们想要的就是比最大的红色值（255）小50的那种颜色，即 $255 - 50 = 205$ 的红色分量。一般地，反色应该是255减去原色。我们需要计算每一个红、绿、蓝分量的反色分量，然后创建一种新的反颜色，并把像素的值设置成这种颜色。

下面是完成这一功能的菜谱，你可以看到它真正达到了目标（如图3.19所示）。



图3.19 底片像



程序16：创建原始图片的底片

```
def negative(image):
    for px in getPixels(image):
        red=getRed(px)
        green=getGreen(px)
        blue=getBlue(px)
        negColor=makeColor( 255-red, 255-green, 255-blue)
        setColor(px,negColor)
```

程序原理

用px表示输入图片的各个像素。对每个像素px，用变量red、green和blue来命名像素颜色的红、绿、蓝分量。用makeColor构造一种新颜色。它的红色分量是255 - red，绿色是255 - green，蓝色是255 - blue。也就是说，新颜色是原始颜色的反色。最后，把像素px的颜色设置成新的反色（negColor）并转向下一个像素。

3.7 转换到灰度

彩色转灰度是一个有趣的菜谱。它很短，理解起来也不难，视觉效果却很不错。这是个很好的例子：通过对像素颜色值的处理，我们能方便地实现强大的功能。

还记得吗？当红、绿、蓝三种颜色分量的值相同时，结果就是灰色。这意味着我们的RGB编码支持256个灰度级。从（0，0，0）（黑色）、（1，1，1）……（100，100，100）直到最后的（255，255，255）（白色）。需要技巧的地方在于计算出应该用哪一级来复制。

我们想要的是一种称为亮度（luminance）的色彩强度（intensity）。事实上，有一种很简单的方法可以计算这个结果：求出三种颜色的平均值。因为有三个分量，所以我们使用的强度公式为

$$\frac{(red + green + blue)}{3}$$

这引出了如下的简单菜谱以及图3.20。



图3.20 彩色图片转换成了灰度图片



程序17：转换成灰度

```
def grayScale(picture):
    for p in getPixels(picture):
        intensity = (getRed(p)+getGreen(p)+getBlue(p))/3
        setColor(p,makeColor(intensity,intensity,intensity))
```

实际上，这里的灰度概念过于简单了。下面的菜谱考虑了人眼如何感知亮度的因素。还记得吗？即使反射的光量相同，我们也觉得蓝色比红色更暗。因此，在计算平均数时，我们可以给蓝色更低的权值（weight），而给红色更高的权值。



程序18：加权法转换为灰度

```
def grayScaleNew(picture):
    for px in getPixels(picture):
        newRed = getRed(px) * 0.299
        newGreen = getGreen(px) * 0.587
        newBlue = getBlue(px) * 0.114
        luminance = newRed+newGreen+newBlue
        setColor(px,makeColor(luminance,luminance,luminance))
```

程序原理

我们用px表示图片的各个像素。然后，根据人们感知红、绿、蓝三种颜色亮度的相关实证研究，我们为三种颜色分配不同的权重。注意， $0.299 + 0.587 + 0.114 = 1.0$ 。最终我们仍会得到0~255之间的一个值。但我们让亮度值较多地来自绿色部分，较少来自红色，更少来自蓝色（我们已经讲过，蓝色是我们感觉最暗的）。我们把三个加权值加在一起得到新的亮度值。最后我们构造了新的颜色值并把它设置为像素px的新颜色。

编程摘要

本章讨论了以下几种数据（或对象）的编码。

图片	图像的编码，通常从JPEG文件中创建出来
像素数组	像素对象的一维数组（序列）。pixels[0]返回位于图片左上角的像素
像素	图片中的一个点。它拥有颜色及与之关联的（x，y）位置。它记录着自己所属的图片，因此改变像素也会改变图片中那个真实的点
颜色	红、绿、蓝三个值的混合，每个值都在0~255之间

图片程序片段

getPixels	接收图片输入，返回图片中像素对象组成的一维数组，数组中首先是来自第一行的像素，然后是来自第二行的像素，以此类推
getPixel	接收一个图片对象和两个位置值x和y（两个数字），返回图片中相应点的像素对象
getWidth	接收图片输入，返回以像素个数表示的图片宽度
getHeight	接收图片输入，返回以像素个数表示的图片高度

(续)

<code>writePictureTo</code>	接收图片和文件名(字符串)作为输入,然后基于JPEG编码将图片写入文件(要保证文件名以“.jpg”结尾,从而操作系统能正确理解它)
<code>getRed, getGreen, getBlue</code>	这三个函数接受一个像素对象,分别返回像素中红色、绿色和蓝色数量(0~255之间)
<code>setRed, setGreen, setBlue</code>	这三个函数接收一个像素对象和一个值(0~255之间),分别将像素的红色、绿色和蓝色数量设置成给定的值
<code>getColor</code>	接收一个像素对象,返回此像素处的颜色对象
<code>setColor</code>	接收一个像素对象和一个颜色对象,为此像素设置颜色
<code>getX, getY</code>	接收一个像素对象,分别返回像素在图片中所处位置的x和y坐标

颜色程序片段

<code>makeColor</code>	接收三项输入,依次为红、绿和蓝色分量,返回一个颜色对象
<code>pickAColor</code>	不需要输入,显示一个颜色选择器,在上面找出你想要的颜色,函数会返回它
<code>makeDarker, makeLighter</code>	这两个函数接受一个颜色对象,分别返回比它稍暗一点或稍亮一点的版本

本章出现了许多有用的常量(constant)。常量是一些具有预定义值的变量。这些值都是颜色,如: `black`、`white`、`blue`、`red`、`green`、`gray`、`darkGray`、`lightGray`、`yellow`、`orange`、`pink`、`magenta`和`cyan`。

习题

3.1 图片概念问题:

- 为什么我们看不到图片中各个位置上的红、绿、蓝点?
- 层次式分解是什么?它适合做什么?
- 亮度是什么?
- 为什么任何颜色分量(红、绿或蓝)的最大值都是255?
- 我们使用的颜色编码是RGB,从表示图片所需的内存数量来看,这意味着什么?我们能够表示的颜色数量有限制吗?RGB能够表示的颜色数目够用吗?

3.2 程序9显然减少了太多红色。试着重写一个只把红色减少10%的版本,然后再试着减少20%。你能找到分别适合用这些不同版本来处理图片吗?注意,你肯定可以反复不断地减少一幅图片中的红色,但你不会愿意重复得次数太多。

3.3 写出消减红色函数(程序9)的蓝色和绿色版本。

3.4 下面的两个函数都等价于增加红色的函数(程序10)。试着测试一下,确保它们是正确的。你更喜欢哪一个?为什么?

```
def increaseRed2(picture):
    for p in getPixels(picture):
        setRed(p, getRed(p)*1.2)

def increaseRed3(picture):
    for p in getPixels(picture):
        redComponent = getRed(p)
        greenComponent = getGreen(p)
```

```

blueComponent = getBlue(p)
newRed=int(redComponent*1.2)
newColor = makeColor
    (newRed,greenComponent,blueComponent)
setColor(p,newColor)

```

- 3.5 如果打开回绕选项并不断增加红色值，最后某些像素会变成光亮的绿色和蓝色。如果使用图片工具查看那些像素，你会发现其红色值非常低。你认为这是怎么回事呢？它们为何变得这么小？回绕的机制的工作原理是什么？
- 3.6 编写一个函数交换两种颜色值，比如交换红色值和蓝色值。
- 3.7 编写一个函数将红色、绿色和蓝色值都变成0，结果会怎样？
- 3.8 编写一个函数将红色、绿色和蓝色值都变成255，结果会怎样？
- 3.9 下面的函数实现了什么功能？

```

def test1 (picture):
    for p in getPixels(picture):
        setRed(p,getRed(p) * 0.3)

```

- 3.10 下面的函数实现了什么功能？

```

def test2 (picture):
    for p in getPixels(picture):
        setBlue(p,getBlue(p) * 1.5)

```

- 3.11 下面的函数实现了什么功能？

```

def test3 (picture):
    for p in getPixels(picture):
        setGreen(p,0)

```

- 3.12 下面的函数实现了什么功能？

```

def test4 (picture):
    for p in getPixels(picture):
        red = getRed(p)
        green = getGreen(p)
        blue = getBlue(p)
        color = makeColor
            (red + 10, green + 10, blue + 10)
        setColor(p,color)

```

- 3.13 下面的函数实现了什么功能？

```

def test5 (picture):
    for p in getPixels(picture):
        red = getRed(p)
        green = getGreen(p)
        blue = getBlue(p)
        color = makeColor
            (red - 20, green - 20, blue - 20)
        setColor(p,color)

```

- 3.14 下面的函数实现了什么功能？

```
def test6 (picture):
    for p in getPixels(picture):
        red = getRed(p)
        green = getGreen(p)
        blue = getBlue(p)
        color = makeColor(blue, red, green)
        setColor(p,color)
```

3.15 下面的函数实现了什么功能?

```
def test7 (picture):
    for p in getPixels(picture):
        red = getRed(p)
        green = getGreen(p)
        blue = getBlue(p)
        color = makeColor
            (red / 2, green / 2, blue / 2)
        setColor(p,color)
```

3.16 编写函数将一幅图片变成灰度图，然后再将它制成底片。

3.17 编写三个函数，一个清除蓝色（程序11），一个清除红色，另一个清除绿色。在实际应用中，哪一个会是最有用的？它们的组合呢？

3.18 重写清除蓝色的函数（程序11），这次把蓝色最大化（即设置成255），而不是清除它。这个函数有用吗？将红色和绿色最大化的版本呢？有用吗？什么情况下有用呢？

3.19 编写函数changeColor，接受三个输入：一幅图片、一个增加或减少的量度以及一个用1、2或3分别表示红、绿、蓝的数字。量度值将在-0.99~0.99之间。

- changeColor(pict, -0.10, 1)应把图片中的红色数量减少10%。
- changeColor(pict, 0.30, 2)应把图片中的绿色数量增加30%。
- changeColor(pict, 0, 3)应当不影响图片中的蓝色数量（红或绿也不影响）。

深入学习

关于视觉的原理以及艺术家是怎样学着处理它，有一本极好的书，那就是Margaret Livingstone编写的《Vision and Art: the Biology of Seeing》[28]。

修改区域中的像素

本章学习目标

本章媒体学习目标：

- 横向或纵向镜像图片。
- 图片组合与拼贴图制作。
- 旋转图片
- 缩放图片

本章计算机科学学习目标：

- 使用嵌套循环寻址矩阵中的元素。
- 循环遍历数组的一部分。
- 开发一些调试策略——特别地，使用print语句研究代码的执行。

4.1 复制像素

在了解像素的具体位置之前，我们基于getPixels能做的图像处理就只有这么多了。例如，如果想针对半幅图片减少红色，那么我们必须有一种方法来指明处理哪些像素。要实现这一目标，我们需要用range来构造自己的for循环。一旦开始这样做，我们就能更精确地控制待处理像素的x和y坐标，于是可以把像素移动到我们想让它去的地方。这是一种非常强大的特性。

如果让一个人拍10下手，那么你怎么知道她做得对呢？你可以数一数她拍的每一下，因此她拍第一下的时候你心里想着1，第二下的时候你想着2，以此类推，直到她停下来。如果她停下来时你的计数为10，那就说明她拍了10下。计数是从多少开始的呢？如果她拍一下之后你想的是1，那就表明在她拍第一下之前计数值实际为0。

for循环与此类似。它用一个索引变量依次接受序列中的每个值。我们已经用getPixels产生过像素序列，实际上，使用Python的range函数，也可以方便地产生数字序列。函数range接受两个输入：一个整数起点和一个不包括在序列范围内的整数终点[⊖]。举几个例子会更清楚。

```
>>> print range(0,3)
[0, 1, 2]
>>> print range(0,10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print range(3,1)
[]
```

你可能感到疑惑，这里的方括号（比如上面第一个例子中的[0, 1, 2]）是什么？它是数组的标记——Python用这种方式输出一列数字，以显示它是一个数组[⊗]。如果用range为for循

⊖ 这可能与你想的不一样，但Python语言就是这样定义的，如果你了解Python的真实机制，不必担心这一点会改变。

⊗ 从技术上讲，range返回的是一个序列(sequence)，序列是一种不同于数组的数据集合，就我们的使用而言，可以认为它就像个数组。

环产生数组，那么我们的变量将遍历结果序列中的每一个数字。

`range`还可以接受一个可选参数：序列中元素之间的增量。

```
>>> print range(0,10,3)
[0, 3, 6, 9]
>>> print range(0,10,2)
[0, 2, 4, 6, 8]
```

由于大多数循环都从0开始（比如，索引一组数据的时候），只给`range`提供单个输入将假定以0为起点。

```
>>> print range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

使用`range`循环遍历像素

要到达坐标为 x 和 y 的像素，我们必须使用两个`for`循环——一个对像素做横向移动（ x ），另一个做纵向移动（ y ），从而得到每个像素（如图4.1所示）。函数`getPixels`在内部就是这样做的，从而简化了一般图像处理的实现。内层循环嵌在外层循环之中，形式上是在它的块中。到这一步，你必须仔细安排代码的缩进，确保各个块的正确排布。

两层循环就像下面这样：

```
for x in range(0,getWidth(picture)):
    for y in range(0,getHeight(picture)):
        pixel=getPixel(picture,x,y)
```

例如，下面的程序与程序14一样，但使用显式的 x 和 y 下标来访问像素。

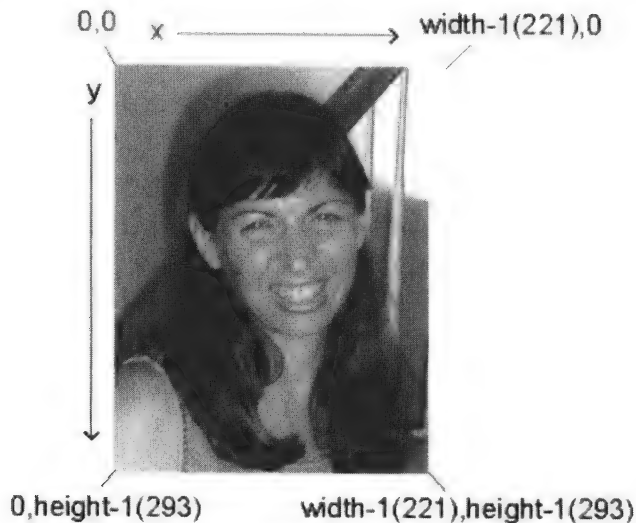


图4.1 图片坐标

程序19：使用嵌套循环亮化图片

```
def lighten(picture):
    for x in range(0,getWidth(picture)):
        for y in range(0,getHeight(picture)):
```



```

px = getPixel(picture,x,y)
color = getColor(px)
color = makeLighter(color)
setColor(px,color)

```

程序原理

我们一起把程序的工作流程走一遍（跟踪一下）。想象我们刚刚执行到语句 `lighten(myPicture)`。

- 1) `def lighten(picture):`，变量 `picture` 成了图片 `myPicture` 的新名字。
- 2) `for x in range(0, getWidth(picture)):`，变量 `x` 接受整数值 0。
- 3) `for y in range(0, getHeight(picture)):`，变量 `y` 接受整数值 0。
- 4) `px = getPixel(picture, x, y)`，变量 `px` 接受位于 (0, 0) 点的像素对象。
- 5) `setColor(px, color)`，我们把 (0, 0) 点的像素对象设置成更亮的颜色。
- 6) `for y in range(0, getHeight(picture)):`，变量 `y` 现在变成 1。换句话说，我们在沿着 `x = 0` 的第一列像素慢慢向下移动。
- 7) `px = getPixel(picture, x, y)`，`px` 变成 (0, 1) 点的像素。
- 8) 加亮该位置的像素。
- 9) `for y in range(0, getHeight(picture)):`，变量 `y` 变成 2。如此循环下去，直到 `y` 等于图片的高度。
- 10) `for x in range(0, getWidth(picture)):`，变量 `x` 接受整数值 1。
- 11) 现在 `y` 再次变成 0，我们沿着 `x = 1` 又处理了下一列。
- 12) 整个过程继续，直到所有像素的颜色加亮。

4.2 图片镜像

让我们从一种有趣的效果开始，它没什么大的用处，有趣而已。我们将沿一条纵轴制作图片的镜像效果。换种说法，可以想象把一面镜子放在图片上，让图片的左半边照在镜中。那就是我们要实现的效果。我们将用多种方法来实现它。

首先，我们把要做的事情通盘考虑一下，而且先把它简化一下。我们将从水平方向上选取一个镜像点 (`mirrorPoint`) —— 图像宽度的一半：`getWidth(picture)/2`。

当图片的宽度为偶数时，我们将把图片的左半边复制到右半边，如图 4.2 所示。这个数组的宽度为 2，因此镜像点为 $2/2 = 1$ 。我们需要把 `x=0, y=0` 复制到 `x=1, y=0`，把 `x=0, y=1` 复制到 `x=1, y=1`。

当图片的宽度为奇数时，我们不复制中间的一列像素，如图 4.2 所示。我们需要把 `x=0, y=0` 复制到 `x=2, y=0`。把 `x=0, y=1` 复制到 `x=2, y=1`。

两种情况下我们的 `x` 和 `y` 都从 0 开始，循环中都是 `x` 介于 $0 \sim \text{mirrorPoint}$ 之间，`y` 介于 0 到图片高度之间。`x` 的值将从 0 一直到 `mirrorPoint` 之前。为实现镜像效果，第一列的颜色应该复制到同一行的最后一列。第二列复制到同一行的倒数第二列，以此类推。因此，当 `x=0` 时，我们将把像素 `x=0, y=0` 的颜色复制到 `x = width-1, y=0`。当 `x=1` 时，我们将把像素 `x=1, y=0` 的颜色复制到 `x = width-2, y=0`。当 `x=2` 时，我们则把像素 `x=2, y=0` 点的颜色复制到 `x = width-3, y=0`。每一次，我们把左边当前 `x`、`y` 值确定的像素的颜色复制到右边图片宽度减去 `x` 值再减 1 的位置。

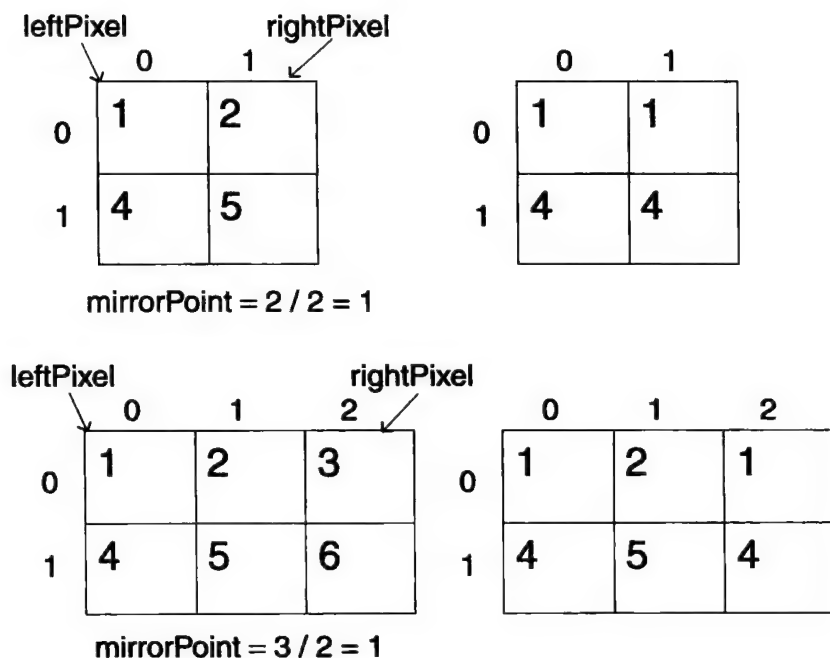


图4.2 以镜像点为轴把像素从左边复制到右边

看一下图4.2，确信我们用这种方法处理了每一个像素。以下是实际的菜谱：



程序20：沿一条纵轴镜像所有像素

```
def mirrorVertical(source):
    mirrorPoint = getWidth(source) / 2
    width = getWidth(source)
    for y in range(0,getHeight(source)):
        for x in range(0,mirrorPoint):
            leftPixel = getPixel(source,x,y)
            rightPixel = getPixel(source,width - x - 1,y)
            color = getColor(leftPixel)
            setColor(rightPixel,color)
```

程序原理

`mirrorVertical`接受一幅源图作为输入。我们采用的是纵向半边镜像，因此`mirrorPoint`等于图片的宽度除以2。我们的处理过程覆盖图片整体高度，因此`y`循环从0开始直到图片的高度。`x`的值从0开始直到`mirrorPoint`（但不包括`mirrorPoint`）。每次循环时，我们都把颜色从左边的一列复制到右边的另一列。

我们将按如下方式使用这个函数（结果显示在图4.3中）。

```
>>> file="C:/ip-book/mediasources/blueMotorcycle.jpg"
>>> print file
C:/ip-book/mediasources/blueMotorcycle.jpg
>>> picture=makePicture(file)
>>> explore(picture)
>>> mirrorVertical(picture)
>>> explore(picture)
```

可以沿横向轴镜像吗？当然可以！



图4.3 原图（左）和沿纵向轴镜像后的图片（右）

**程序21：横向镜像像素，从上到下**

```
def mirrorHorizontal(source):
    mirrorPoint = getHeight(source) / 2
    height = getHeight(source)
    for x in range(0,getWidth(source)):
        for y in range(0,mirrorPoint):
            topPixel = getPixel(source,x,y)
            bottomPixel = getPixel(source,x,height - y - 1)
            color = getColor(topPixel)
            setColor(bottomPixel,color)
```

上面的菜谱把图片上面的部分复制到下面（如图4.4所示）。可以看到，我们从当前 x 、 y 确定的 $topPixel$ 中取得颜色，这一点肯定在 $mirrorPoint$ 以上，因为较小的 y 值更靠近图的上方。如果要从下往上复制，把 $topPixel$ 和 $bottomPixel$ 交换一下即可。

**程序22：横向镜像像素，从下到上**

```
def mirrorBotTop(source):
    mirrorPoint = getHeight(source) / 2
    height = getHeight(source)
    for x in range(0,getWidth(source)):
        for y in range(0,mirrorPoint):
            topPixel = getPixel(source,x,y)
            bottomPixel = getPixel(source,x,height - y - 1)
            color = getColor(bottomPixel)
            setColor(topPixel,color)
```

有用的镜像

虽然镜像操作多用于制作有趣效果，但偶尔它也会用于更严肃的目的（但仍然有趣）。Mark在雅典的古安哥拉遗址拍了赫菲斯托斯神庙的一张相片。神庙的山墙有损毁。他在想，能否将完好的部分镜像到损毁的部分，从而“修复”它呢？

为找到镜像点的坐标，我们可以用JES的图片工具来浏览图片。

```
>>> file = "C:/ip-book/mediasources/temple.jpg"
>>> templeP = makePicture(file)
>>> explore(templeP)
```



图4.4 横向镜像，从上到下（左）和从下到上（右）



图4.5 摄于雅典古安哥拉遗址的赫菲斯托斯神庙

为找出需要镜像的范围和镜像轴的位置，Mark浏览了这幅图片（如图4.6所示）。他编写的修复函数列在下面，最终的图片如图4.7所示——效果非常不错！当然，我们可以辨别出它是经过数字化处理的。比如，检查一下阴影，你会发现只有太阳同时出现在左右两边时才会这样。

我们将使用函数`getMediaPath(baseName)`，这是个工具函数（utility function）。当需要处理同一目录下的多个媒体文件，又不想拼出完整的路径名时，它特别有用。你只要记住首先使用`setMediaPath()`。`getMediaPath`所做的就是将`setMediaPath`设置的路径名添加到输入文件名之前。

```
>>> setMediaPath()
New media folder: 'C:\\ip-book\\mediasources\\'
>>> getMediaPath("barbara.jpg")
'C:\\ip-book\\mediasources\\barbara.jpg'
>>> barb=makePicture(getMediaPath("barbara.jpg"))
```

程序23：镜像赫菲斯托斯神庙

```
def mirrorTemple():
    source = makePicture(getMediaPath("temple.jpg"))
    mirrorPoint = 276
    for x in range(13, mirrorPoint):
```



```

for y in range(27,97):
    pleft = getPixel(source,x,y)
    pright = getPixel(source,mirrorPoint + mirrorPoint - 1 - x,y)
    setColor(pright,getColor(pleft))
show(source)
return source

```



常见bug：首先设置媒体文件夹

如果想在代码中使用getMediaPath(baseName)，需要首先执行setMediaPath()。

程序原理

我们先在JES中浏览图片，从而找到了mirrorPoint(276)。实际上，我们不需要从0开始一直复制到276，因为神庙的边缘在x坐标为13的地方。

在这份菜谱中，我们使用了getMediaPath。getMediaPath(baseName)函数提供了一种省略方法。如果媒体文件存放在某个地方，你希望只用基本文件名来引用它，那么就可以使用getMediaPath(baseName)，实际由它来产生完整的路径名。但是，你必须首先调用setMediaPath()。setMediaPath()函数让你选择一个保存媒体的位置（目录）。它告诉getMediaPath(baseName)如何构造文件路径。后者返回由媒体目录后与基本文件名构成的路径。

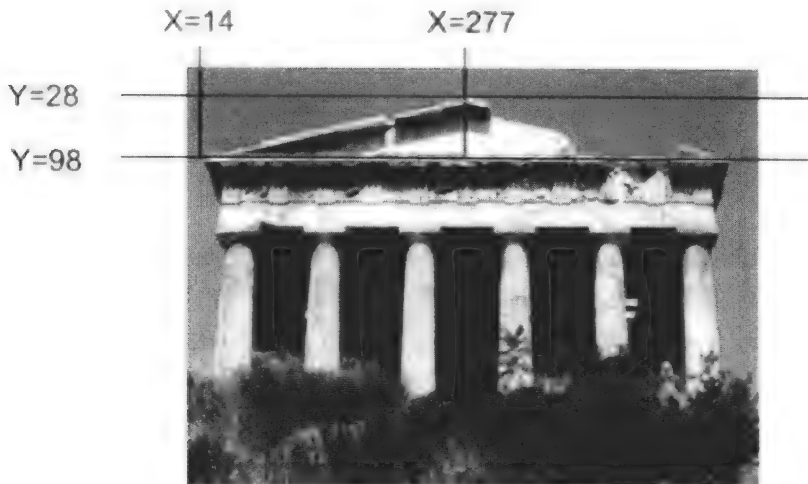


图4.6 需要镜像的坐标位置

这份菜谱也是我们编写的第一份显式返回一个值的菜谱。关键字return设置了函数为输出而提供的值。在mirrorTemple()中，返回值就是图片对象source，它保存修复后的神庙。如果我们通过fixedTemple = mirrorTemple()来调用这个函数，那么名字fixedTemple将代表mirrorTemple()返回的图片。



常见bug：关键字return总在最后

关键字return指出了函数的返回值，但它同时还有终止函数的作用。一种常见的bug就是在return语句之后仍试图执行print语句或show函数，但发现没有效果。一旦执行了return，函数中就不会再有语句执行了。



图4.7 处理后的神庙

为什么我们要返回修复后的神庙图片呢？为什么之前我们从来没有从函数中返回东西呢？我们之前编写的函数都直接处理输入对象——这称为基于副作用（side effect）的计算。在这个例子中我们不能这么做，因为`mirrorTemple()`没有输入。关于何时用`return`，基本的法则是：如果函数中创建了外界感兴趣的对象，那么你就需要返回它；否则，函数结束时它就消失了。图片对象`source`是在`mirrorTemple()`函数内部（使用`makePicture()`）创建的，它只存在于函数内部。

为什么要返回一样东西？为了以后使用。你能想象用修复的神庙图片做一件事吗？将它收集到拼贴图中？改变它的颜色？你应该返回这个对象，从而可以之后再作选择。

神庙程序是个很好的例子，针对它我们可以提问许多问题。如果真正理解了它，你就能回答这样的问题：“哪个像素是函数中第一个被镜像的？”“函数一共复制了多少像素？”你应该做到只需在大脑中过一遍程序就能想出这些答案——把自己当成计算机，在大脑中执行程序。

如果这太难，可以插入一些`print`语句，就像下面这样：

```
def mirrorTemple():
    source = makePicture(getMediaPath("temple.jpg"))
    mirrorPoint = 276
    for x in range(13, mirrorPoint):
        for y in range(27, 97):
            print "Copying color from", x, y, "to", mirrorPoint + mirrorPoint - 1 - x, y
            pleft = getPixel(source, x, y)
            pright = getPixel(source, mirrorPoint + mirrorPoint - 1 - x, y)
            setColor(pright, getColor(pleft))
    show(source)
    return source
```

这个版本要花很长时间才能运行结束。运行一小段后可以点击`Stop`按钮，因为我们只关心前面几个像素。以下是我们得到的：

```
>>> p2=mirrorTemple()
Copying color from 13 27 to 538 27
```

```
Copying color from 13 28 to 538 28
Copying color from 13 29 to 538 29
.
```

它将像素 (13, 27) 复制到 (538, 27), 其中538是通过`mirrorPoint(276)`加上`mirrorPoint (552)`再减1 (551), 然后减去`x (551 - 13 = 538)`得到的。

我们总共处理了多少像素? 这个答案也可以让计算机给出。在循环开始前把一个计数值设置成0, 然后每复制一个像素就把计数值加1。

```
def mirrorTemple():
    source = makePicture(getMediaPath("temple.jpg"))
    mirrorPoint = 276
    count = 0
    for x in range(13, mirrorPoint):
        for y in range(27, 97):
            pleft = getPixel(source, x, y)
            pright = getPixel(source, mirrorPoint + mirrorPoint - 1 - x, y)
            setColor(pright, getColor(pleft))
            count = count + 1
    show(source)
    print "We copied", count, "pixels"
    return source
```

这段程序输出了 “We copied 18410 pixels.” (我们复制了18 410个像素)。这个数字是怎么来的呢? 我们复制了70行 (`y`介于27~97) 263列 (`x`介于13~276) 像素, $70 \times 263 = 18\,410$ 。

4.3 复制和转换图片

在复制图片像素的过程中, 可以创建全新的图片。我们将跟踪一幅源图片和一幅目标图片, 从源图片中取得像素并设置目标图片中的像素。实际上, 我们并不复制像素——只是让目标图片中的像素拥有与源图片像素一样的颜色。复制像素需要我们跟踪多个坐标变量: 源图片中的 (`x`, `y`) 位置和目标图片中的 (`x`, `y`) 位置。

关于复制像素, 令人兴奋的一点是: 只需对处理坐标变量的方式做一点小小改变, 就可以做到不仅复制了图像, 而且转换了图像。这一节将讨论图片的复制、裁剪、旋转和缩放。

我们的目标图片将是MEDIASOURCES目录中纸张大小的JPEG文件, 即7 in × 9.5 in, 在9 in × 11.5 in的信笺纸张上留出1 in的页边距打印, 它正合适。

```
>>> paperFile=getMediaPath("7inx95in.jpg")
>>> paperPicture=makePicture(paperFile)
>>> print getWidth(paperPicture)
504
>>> print getHeight(paperPicture)
684
```

4.3.1 复制

要把图片从一个对象复制到另一个对象, 我们只需确保同时递增`sourceX`和`targetX`变量 (`X`轴的源坐标和目标坐标变量) 以及`sourceY`变量和`targetY`变量。可以用一个`for`循环, 但它只能递增一个变量。我们必须保证在`for`循环递增变量的同时 (在足够近的地方) 使用某种表达式来递增另一个不在`for`语句中的变量。最后我们采用的方法是: 在`for`循环即将开始之前

设置索引变量的初始值，然后在循环底部增加索引变量。

下面是把Barbara的相片复制到画布（canvas）的菜谱。



程序24：将图片复制到画布

```
def copyBarb():
    # 设置源图片和目标图片
    barbf=getMediaPath("barbara.jpg")
    barb = makePicture(barbf)
    canvasf = getMediaPath("7inX95in.jpg")
    canvas = makePicture(canvasf)
    # 这里实际执行复制
    targetX = 0
    for sourceX in range(0,getWidth(barb)):
        targetY = 0
        for sourceY in range(0,getHeight(barb)):
            color = getColor(getPixel(barb,sourceX,sourceY))
            setColor(getPixel(canvas,targetX,targetY), color)
            targetY = targetY + 1
        targetX = targetX + 1
    show(barb)
    show(canvas)
    return canvas
```

计算机科学思想：注释有益

在程序24中，你可以看到一些以“#”开头的行。这一符号告诉Python“忽略本行剩下的部分”。这有什么好处呢？有了它，你可以在程序中放置一些供人阅读而不是供计算机阅读的消息——这些消息可以解释程序的机制、程序执行的部分，以及那样做的原因。别忘了程序是供人阅读的，而不是供计算机阅读的。注释使程序更适于人们阅读。

程序原理

这个程序将Barbara的相片复制到了画布上（如图4.8所示）。

- 开始的几行设置了源图片（barb）和目标图片（canvas）。
- 接下来的循环管理X轴的坐标变量：用于源图片的sourceX和用于目标图片的targetX。

以下是循环的关键部分：

```
targetX = 0
for sourceX in range(0,getWidth(barb)):
    # 这里是Y循环
    targetX = targetX + 1
```

根据语句的排列情况，从Y轴循环的角度来看，targetX和sourceX总是同时递增的。targetX变成0之后紧接着for循环中的sourceX也变成了0。在for循环的末尾，targetX递增1，然后循环重新开始，通过for语句，sourceX也递增1。

递增targetX的语句看上去有些奇怪。targetX = targetX + 1不是一条数学表达式（在数学上这是不可能成立的）。实际上它是发给计算机的指令。它说的是“让targetX的值成为（等号右边）targetX的当前值加1”。

- X轴循环之内是Y轴坐标变量的循环。它有着与X轴循环类似的结构，因为它的目标是以完全相同的方式让targetY和sourceY保持同步。


```

targetY = 0
for sourceY in range(0,getHeight(barb)):
    color = getColor(getPixel(barb,sourceX,sourceY))
    setColor(getPixel(canvas,targetX,targetY), color)
    targetY = targetY + 1

```



图4.8 将图片复制到画布中 (1)

实际上，我们是在Y循环内部从源图片获得颜色并将目标图片中的对应像素设置成相同颜色。

计算机科学思想：复制与引用

当我们从源图片向目标图片复制颜色的时候，目标图片只含有颜色信息。它对源图片一无所知。如果修改了源图片，那么目标图片不会改变。从计算机科学的意义上，也可以使用引用（reference）。引用是指向另一个对象的指针。如果目标图片含有指向源图片的指针，那么这时如果修改了源图片，目标图片也会改变。

事实上，我们也完全可以把目标变量放在for循环语句中，而单独设置源变量。下面的菜谱完成的动作与程序24是一样的。

程序25：另一种将图片复制到画布中的方法

```

def copyBarb2():
    # 设置源图片和目标图片
    barbf=getMediaPath("barbara.jpg")
    barb = makePicture(barbf)
    canvasf = getMediaPath("7inX95in.jpg")
    canvas = makePicture(canvasf)
    # 这里实际执行复制
    sourceX = 0
    for targetX in range(0,getWidth(barb)):
        sourceY = 0
        for targetY in range(0,getHeight(barb)):
            color = getColor(getPixel(barb,sourceX,sourceY))
            setColor(getPixel(canvas,targetX,targetY), color)
            sourceY = sourceY + 1
        sourceX = sourceX + 1

```

```

show(barb)
show(canvas)
return canvas

```

当然，我们未必非得把源图片的(0, 0)点复制到目标图片的(0, 0)点。我们完全可以复制到画布的其他位置。需要做的只是改变目标X和Y坐标的起点。其他的完全一样（如图4.9所示）。



图4.9 将图片复制到画布中间



程序26：复制到画布的其他位置

```

def copyBarbMidway():
    # 设置源图片和目标图片
    barbf=getMediaPath("barbara.jpg")
    barb = makePicture(barbf)
    canvasf = getMediaPath("7inX95in.jpg")
    canvas = makePicture(canvasf)
    # 这里实际执行复制
    sourceX = 0
    for targetX in range(0,getWidth(barb)):
        sourceY = 0
        for targetY in range(0,getHeight(barb)):
            color = getColor(getPixel(barb,sourceX,sourceY))
            setColor(getPixel(canvas,targetX,targetY), color)
            sourceY = sourceY + 1
        sourceX = sourceX + 1
    show(barb)
    show(canvas)
    return canvas

```

类似地，我们也不一定要复制整幅图片。裁剪（crop）就是从整幅图片中取出一部分。从数字上讲，那不过是改变一下起点和终点的坐标。要从图片中仅取走Barb的脸庞，我们只需找到她的脸庞所在的位置，然后分别在sourceX和sourceY的维度上使用它们（如图4.10所示）。通过浏览图片，我们可以找出坐标。脸的位置在(45, 25) ~ (200, 200)之间。



图4.10 复制图片的一部分到画布中

**程序27：在画布上裁剪图片**

```
def copyBarbsFace():
    # 设置源图片和目标图片
    barbf=getMediaPath("barbara.jpg")
    barb = makePicture(barbf)
    canvasf = getMediaPath("7inX95in.jpg")
    canvas = makePicture(canvasf)
    # 这里实际执行复制
    targetX = 100
    for sourceX in range(45,200):
        targetY = 100
        for sourceY in range(25,200):
            color = getColor(getPixel(barb,sourceX,sourceY))
            setColor(getPixel(canvas,targetX,targetY), color)
            targetY = targetY + 1
        targetX = targetX + 1
    show(barb)
    show(canvas)
    return canvas
```

程序原理

这个程序与前一个程序的唯一区别是源坐标的范围。我们只想要 x 在45~200之间的像素，因此这两个数字就是用于 $sourceX$ 的 $range$ 的输入。我们只想要 y 在25~200之间的像素，因此这两个数字就是用于 $sourceY$ 的 $range$ 的输入，剩下的都一样。

我们仍然可以把 for 循环语句中的变量与显式递增的变量交换一下。目标范围的计算却有点儿复杂，我们想从目标起点（100，100）处开始把像素复制过来，图片的宽度是200 - 45，高度是200 - 25，以下是完整的菜谱。

**程序28：使用不同的方法复制脸庞到画布**

```
def copyBarbsFace2():
    # 设置源图片和目标图片
    barbf=getMediaPath("barbara.jpg")
```

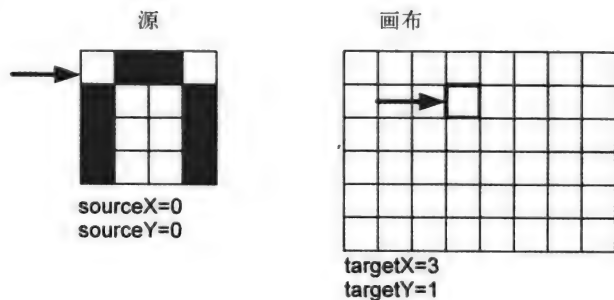
```

barb = makePicture(barbf)
canvasf = getMediaPath("7inX95in.jpg")
canvas = makePicture(canvasf)
# 这里实际执行复制
sourceX = 45
for targetX in range(100,100+(200-45)):
    sourceY = 25
    for targetY in range(100,100+(200-25)):
        color = getColor(getPixel(barb,sourceX,sourceY))
        setColor(getPixel(canvas,targetX,targetY), color)
        sourceY = sourceY + 1
    sourceX = sourceX + 1
show(barb)
show(canvas)
return canvas

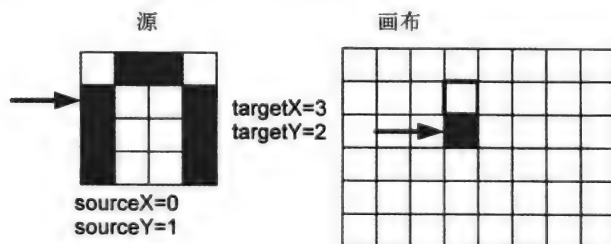
```

程序原理

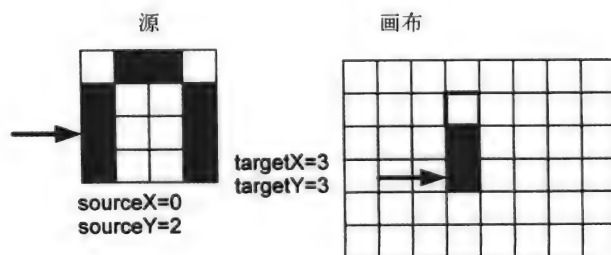
让我们通过一个小例子来弄清楚上面的复制菜谱做了什么。我们从一个源和一个目标开始，把像素逐个从源复制到目标。



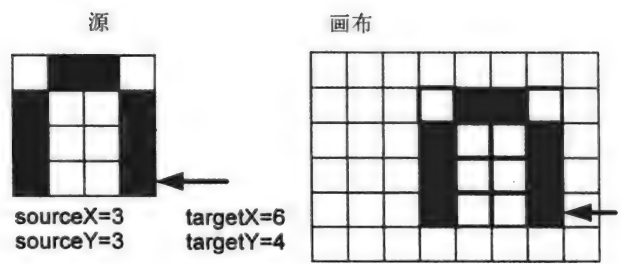
然后，我们递增sourceY和targetY，再次复制。



我们沿着列继续向下，递增两个Y坐标变量。



复制好这一列之后，我们递增X坐标变量，转到下一列，直到复制完每个像素。



4.3.2 制作拼贴图

下面是两幅花朵的图片（如图4.11所示），每幅图片的宽度和高度都是100像素。让我们组合不同的效果来制作不同的花朵，然后用它们组成一张拼贴图（collage）。我们将把这些花朵全部复制到一幅空白图像640x480.jpg中。实际上，我们只需把像素的颜色复制到正确的位置上即可。

以下就是我们制作拼贴图（如图4.12所示）的方法：

```
>>> flowers = createCollage()
```

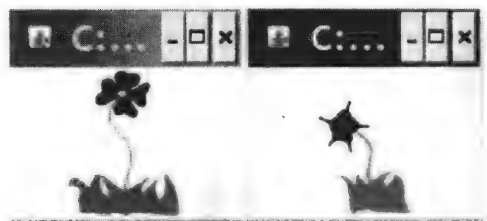


图4.11 拼贴图中使用的花朵图片

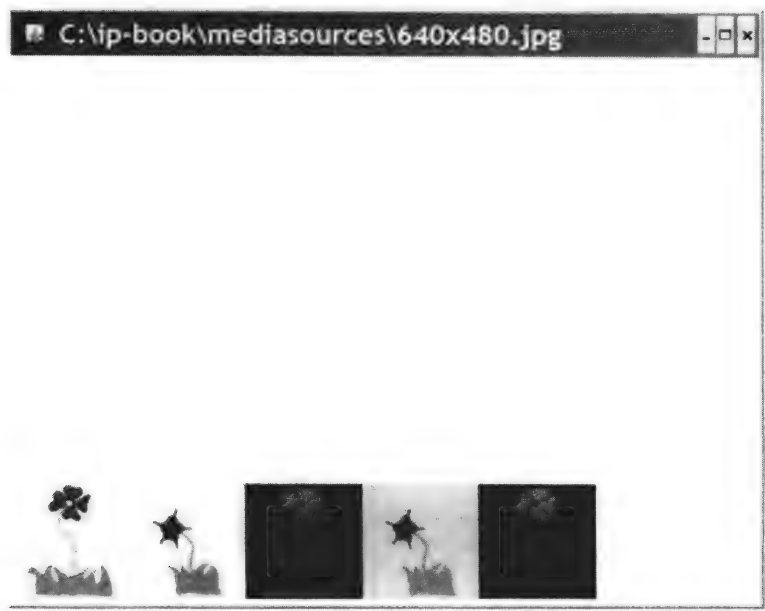


图4.12 花朵拼贴图



常见bug：引用的函数必须位于文件中

这个程序使用了我们以前编写的函数。为使程序正常工作，这些函数必须复制到createCollage函数所在的文件中。（后面我们会看到如何用import引用其他文件中的函数。）



程序29：制作拼贴图

```
def createCollage():
    flower1=makePicture(getMediaPath("flower1.jpg"))
    print flower1
    flower2=makePicture(getMediaPath("flower2.jpg"))
    print flower2
    canvas=makePicture(getMediaPath("640x480.jpg"))
    print canvas
    # 第一幅图片，从左边开始
    targetX=0
    for sourceX in range(0,getWidth(flower1)):
        targetY=getHeight(canvas)-getHeight(flower1)-5
        for sourceY in range(0,getHeight(flower1)):
            px=getPixel(flower1,sourceX,sourceY)
            cx=getPixel(canvas,targetX,targetY)
            setColor(cx,getColor(px))
            targetY=targetY + 1
            targetX=targetX + 1
    # 第二幅图片，往右100个像素
    targetX=100
    for sourceX in range(0,getWidth(flower2)):
        targetY=getHeight(canvas)-getHeight(flower2)-5
        for sourceY in range(0,getHeight(flower2)):
            px=getPixel(flower2,sourceX,sourceY)
            cx=getPixel(canvas,targetX,targetY)
            setColor(cx,getColor(px))
            targetY=targetY + 1
            targetX=targetX + 1
    # 第三幅图片，flower1颜色反转
    negative(flower1)
    targetX=200
    for sourceX in range(0,getWidth(flower1)):
        targetY=getHeight(canvas)-getHeight(flower1)-5
        for sourceY in range(0,getHeight(flower1)):
            px=getPixel(flower1,sourceX,sourceY)
            cx=getPixel(canvas,targetX,targetY)
            setColor(cx,getColor(px))
            targetY=targetY + 1
            targetX=targetX + 1
    # 第四幅图片，没有蓝色的flower2
    clearBlue(flower2)
    targetX=300
    for sourceX in range(0,getWidth(flower2)):
        targetY=getHeight(canvas)-getHeight(flower2)-5
        for sourceY in range(0,getHeight(flower2)):
            px=getPixel(flower2,sourceX,sourceY)
            cx=getPixel(canvas,targetX,targetY)
            setColor(cx,getColor(px))
            targetY=targetY + 1
            targetX=targetX + 1
    # 第五幅图片，颜色反转后又减少红色的flower1
    decreaseRed(flower1)
    targetX=400
```

```

for sourceX in range(0,getWidth(flower1)):
    targetY=getHeight(canvas)-getHeight(flower1)-5
    for sourceY in range(0,getHeight(flower1)):
        px=getPixel(flower1,sourceX,sourceY)
        cx=getPixel(canvas,targetX,targetY)
        setColor(cx,getColor(px))
        targetY=targetY + 1
        targetX=targetX + 1
show(canvas)
return(canvas)

```

程序原理

这个程序虽然看起来很长，但实际上与我们多次见过的复制循环是一样的，只不过这次的复制是一个接着一个。

- 首先创建要复制到canvas中的图片对象flower1和flower2。
- 第1朵花只是flower1的复制，它位于画布的最左边。我们想让花朵的底部离开边缘5个像素，因此targetY的初始值为画布的高度减去花的高度再减去5。随着targetY不断递增，它将逐渐向下靠近底部。它递增的次数等于花朵图片的像素高度（见sourceY循环），因此targetY的最大值等于画布的高度减5。
- 接着把第二幅图片复制进来，从targetX等于100开始向右复制，实际是同样的循环。
- 现在反转flower1的颜色，然后把它复制进来，画布中的图像进一步向右移（targetX现在从300开始）。
- 然后清除了flower2中的蓝色并把它复制到画布中更靠右的位置。
- 第5朵花把（在第三组循环中）已被反转过颜色的flower1中的红色减少。
- 然后用show显示了画布并用return返回了它。我们需要返回画布，因为它是在拼贴图函数内部创建的。如果不返回它，当函数结束，函数的上下文也结束时它就消失了。

4.3.3 通用复制

制作拼贴图的代码特别长，而且充斥着重复代码。每次向目标画布复制一幅图片时，我们都要计算targetY并设置targetX。然后我们循环遍历源图中的所有像素并全部复制到目标图片。有什么办法能让它短一些呢？如果创建一个通用的复制函数，让它接受待复制图片、目标图片，并指定复制时目标图片中的起始坐标，这样会不会好呢？



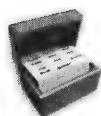
程序30：通用的复制函数

```

def copy(source, target, targX, targY):
    targetX = targX
    for sourceX in range(0,getWidth(source)):
        targetY = targY
        for sourceY in range(0,getHeight(source)):
            px=getPixel(source,sourceX,sourceY)
            tx=getPixel(target,targetX,targetY)
            setColor(tx,getColor(px))
            targetY=targetY + 1
            targetX=targetX + 1

```

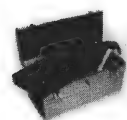
现在，有了这个新的通用复制函数，我们可以用它来重写拼贴图函数。



程序31：使用通用复制函数改善过的拼贴图程序

```
def createCollage2():
    flower1=makePicture(getMediaPath("flower1.jpg"))
    print flower1
    flower2=makePicture(getMediaPath("flower2.jpg"))
    print flower2
    canvas=makePicture(getMediaPath("640x480.jpg"))
    print canvas
    # 第一幅图片，从左边开始
    copy(flower1,canvas,0,getHeight(canvas)-getHeight(flower1)-5)
    # 第二幅图片，往右100个像素
    copy(flower2,canvas,100,getHeight(canvas)-getHeight(flower2)-5)
    # 第三幅图片，flower1颜色反转
    negative(flower1)
    copy(flower1,canvas,200,getHeight(canvas)-getHeight(flower1)-5)
    # 第四幅图片，没有蓝色的flower2
    clearBlue(flower2)
    copy(flower2,canvas,300,getHeight(canvas)-getHeight(flower2)-5)
    # 第五幅图片，颜色反转后又减少红色的flower1
    decreaseRed(flower1)
    copy(flower1,canvas,400,getHeight(canvas)-getHeight(flower2)-5)
    return canvas
```

现在，制作拼贴图的代码阅读、修改和理解起来就容易多了。编写接受参数的函数能使它们更易于重用。在函数中多次重复代码还会带来其他问题，如果重复代码含有错误时，你必须在多个地方改正错误，而不是只改一个地方。



实践技巧：不要复制函数，要重用函数

试着编写接受参数的可重用函数。要抵制多处复制代码的习惯，因为它会使代码更长，而且制造出难以改正的错误。

4.3.4 旋转

其他方面都一样，但以不同的方式使用坐标变量或以不同的方式递增它们，我们还可以实现图像的转换（Rotation）。我们来把Barb向左旋转90°——至少看起来是这样。实际上是沿着对角线翻转图像，我们将通过交换目标图片中的X和Y变量来实现——以相同的方式递增它们，但将X用于Y，Y用于X（如图4.13所示）。



图4.13 将图片复制到画布中（2）



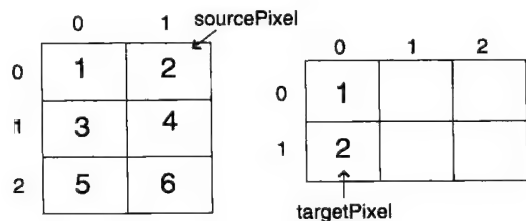
程序32：旋转（翻转）图片

```
def copyBarbSideways():
    # 设置源图片和目标图片
    barbf=getMediaPath("barbara.jpg")
    barb = makePicture(barbf)
    canvasf = getMediaPath("7inX95in.jpg")
    canvas = makePicture(canvasf)
    # 这里实际执行复制
    targetX = 0
    for sourceX in range(0,getWidth(barb)):
        targetY = 0
        for sourceY in range(0,getHeight(barb)):
            color = getColor(getPixel(barb,sourceX,sourceY))
            setColor(getPixel(canvas,targetY,targetX), color)
            targetY = targetY + 1
        targetX = targetX + 1
    show(barb)
    show(canvas)
    return canvas
```

程序原理

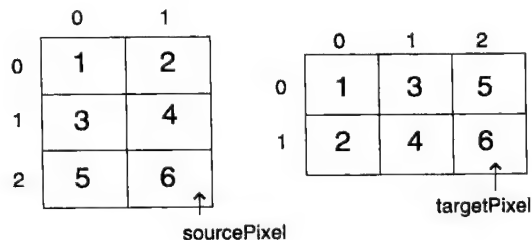
旋转（沿对角线翻转）同样从设置源和目标开始，甚至使用的变量值都是一样的，但因为以不同的方式使用X和Y，所以我们得到了不同的效果。为使这个问题便于理解，我们用一个数字小矩阵说明一下。

现在，随着Y变量的递增，我们在沿着源数组向下移动，而在目标数组中却是横向移动的。



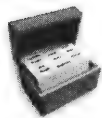
```
sourceX = 0
sourceY = 1
targetX = sourceY = 1
targetY = sourceX = 0
```

结束时，我们完成了同样的复制，效果却截然不同。



```
sourceX = 1
sourceY = 2
targetX = sourceY = 2
targetY = sourceX = 1
```

怎样实际做到90°的旋转呢？我们需要考虑一下，你想让每个像素到哪里去？下面的程序实际完成了图片的90°旋转。关键的区别在于setColor函数的调用。注意，我们交换了前一个例子中的x和y坐标，但我们为y使用了一个公式： $width - targetX - 1$ 。试着跟踪一下程序，确信它实现了真正的旋转。



程序33：旋转图片

```
def rotateBarbSideways():
    # 设置源图片和目标图片
    barbf=getMediaPath("barbara.jpg")
    barb = makePicture(barbf)
    canvasf = getMediaPath("7inX95in.jpg")
    canvas = makePicture(canvasf)
    # 这里实际执行复制
    targetX = 0
    width = getWidth(barb)
    for sourceX in range(0,getWidth(barb)):
        targetY = 0
        for sourceY in range(0,getHeight(barb)):
            color = getColor(getPixel(barb,sourceX,sourceY))
            setColor(getPixel(canvas,targetY,width - targetX - 1), color)
            targetY = targetY + 1
        targetX = targetX + 1
    show(barb)
    show(canvas)
    return canvas
```

4.3.5 缩放

缩放是一种极其常见的图片转换。“缩”是指让图片更小，“放”是指让图片更大。将100万像素或300万像素的图片缩成更小的尺寸使之更易于放到Web上是一种常见的做法。更小的图片占用更少的磁盘空间、更少的网络带宽，从而可以更方便、更快速地下载。

缩放图片需要使用采样（sampling）技术，这种技术我们后面讲声音时还会用到。对于图片缩小，我们从源图片复制像素到目标图片时，将采用隔一个复制一个的方法。对于图片放大，我们将采用每个像素取用两次的方法。

图片缩小相对简单，只需每次把源X、Y变量加2，而不是加1。因为想占用一半的地方，我们把空间的尺寸除以2——宽度将变成 $(200 - 45) / 2$ ，高度则是 $(200 - 25) / 2$ ，目标位置仍然从（100，100）开始。结果就得到画布上的一张小脸（如图4.14所示）。



图4.14 缩小图片



程序34：缩小图片

```
def copyBarbsFaceSmaller():
    # 设置源图片和目标图片
    barbf=getMediaPath("barbara.jpg")
    barb = makePicture(barbf)
    canvasf = getMediaPath("7inX95in.jpg")
    canvas = makePicture(canvasf)
    # 这里实际执行复制
    sourceX = 45
    for targetX in range(100,100+((200-45)/2)):
        sourceY = 25
        for targetY in range(100,100+((200-25)/2)):
            color = getColor(getPixel(barb,sourceX,sourceY))
            setColor(getPixel(canvas,targetX,targetY), color)
            sourceY = sourceY + 2
            sourceX = sourceX + 2
    show(barb)
    show(canvas)
    return canvas
```

程序原理

- 第一步创建图片对象。barb是源，canvas是我们创作Barb的地方。
- Barb的脸在矩形范围（45，25）～（200，200）之间，这意味着sourceX从45开始，sourceY从25开始。我们没有为源坐标指定范围终点，因为它也受制于针对目标坐标的for循环。
- 我们让targetX和targetY都从100开始。范围的终点是什么呢？我们要得到Barb完整的Barb脸庞。脸的宽度是 $200 - 45$ （x坐标最大值减最小值）。因为想把Barb的脸缩小一半（两个方向都缩小），我们每隔一个像素跳过一个。那意味着最终图片的宽度只有源宽度的一半： $(200-45)/2$ 。如果让targetX从100开始，那么范围的终点就是100加上 $(200-45)/2$ 。targetY与之类似。
- 因为想在源图片中每隔一个像素跳过一个，循环过程中我们每次都把sourceX和sourceY加2。

放大图片更复杂一些。我们想把每个像素取用两次。每次我们将把源坐标变量增加0.5。我们无法引用坐标为1.5的像素，但如果引用`int(1.5)`（取整函数），我们便又得到了1，这样就不会有问题。序列1、1.5、2、2.5…将变成1、1、2、2…。结果就是一幅更大的图片（如图4.15所示）。



图4.15 放大图片



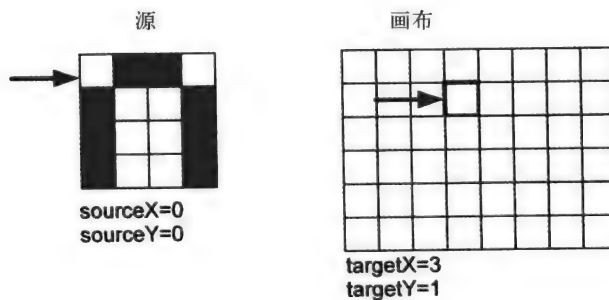
程序35：放大图片

```
def copyBarbsFaceLarger():
    # 设置源图片和目标图片
    barbf=getMediaPath("barbara.jpg")
    barb = makePicture(barbf)
    canvasf = getMediaPath("7inX95in.jpg")
    canvas = makePicture(canvasf)
    # 这里实际执行复制
    sourceX = 45
    for targetX in range(100,100+((200-45)*2)):
        sourceY = 25
        for targetY in range(100,100+((200-25)*2)):
            color = getColor(getPixel(barb,int(sourceX),int(sourceY)))
            setColor(getPixel(canvas,targetX,targetY), color)
            sourceY = sourceY + 0.5
            sourceX = sourceX + 0.5
    show(barb)
    show(canvas)
    return canvas
```

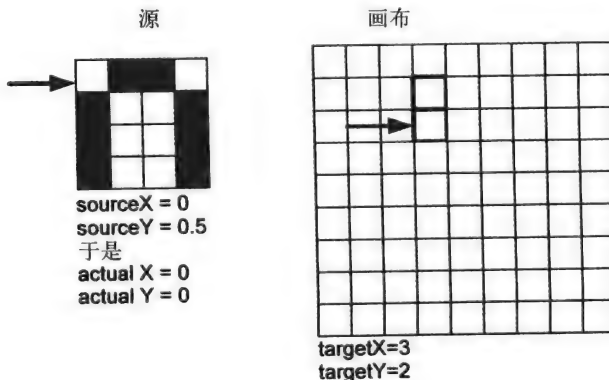
你可能考虑将图片放大到特定尺寸，而不是一直使用画布图片。有一个函数叫 `makeEmptyPicture()`，它能以期望的宽度和高度（皆以像素数指定）创建图片。`makeEmptyPicture(640, 480)`将创建640像素宽、480像素高的图片对象——就像画布一样。

程序原理

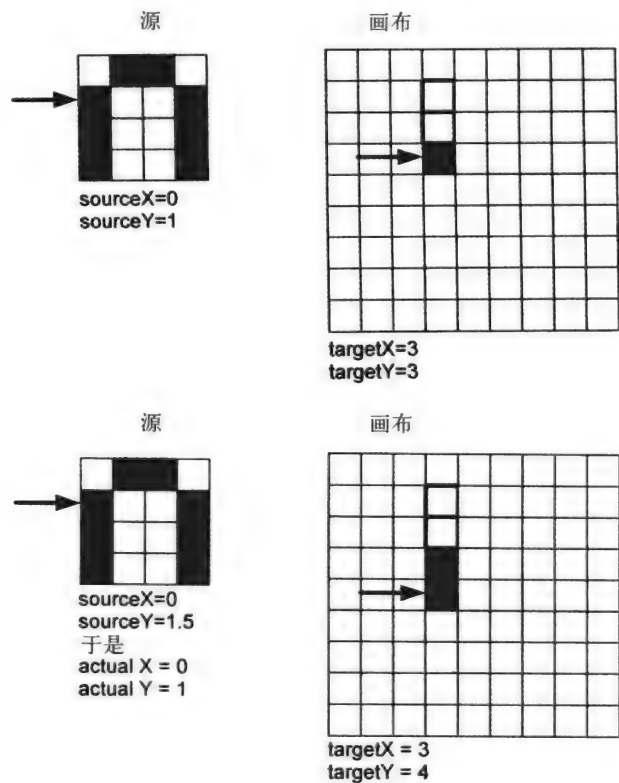
起始位置与之前的复制过程一样。



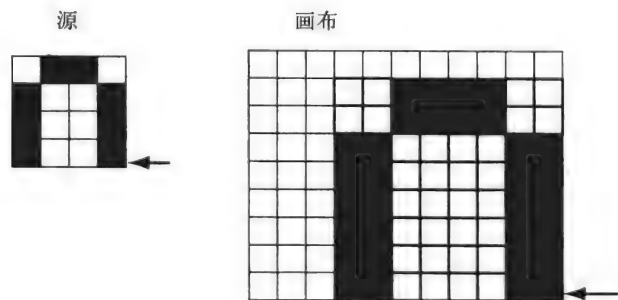
当 `sourceY` 增加 0.5，我们实际引用了源图中的同一个像素，但在目标图中却移向了下一个像素。



现在，当sourceY再次增加0.5，我们移向了下一个像素，这个像素最终也会复制两次。



当我们在目标图中移向新的一行时，源图中我们还处在同一行上，最后，我们把每个像素都复制了两次，横向、纵向上都是这样。最终图片在两个方向上都增大了一倍，结果图片的面积就成了原来的4倍。注意最终图片的质量降低了一些：锯齿比原先的图更明显。



编程摘要

range	创建一连串数字的函数。对创建数组或矩阵的下标有用
setMediaPath()	使用文件选择器选取媒体目录
setMediaPath(directory)	指定媒体目录
getMediaPath(baseName)	接受一个基本文件名，返回此文件的完整路径（假定它位于媒体目录中）
makeEmptyPicture(width, height)	接受宽度和高度，返回指定大小的空（全白）图片

习题

```
4.1 def newFunction(a, b, c):
    print a
    list1 = range(0,4)
    value = 0
    for x in list1:
        print b
        value = value +1
    print c
    print value
```

如果输入newFunction("I", "you", "walrus")来调用上面的函数，结果会输出什么？

4.2 下列菜谱中有哪些接受一幅图片并去除所有蓝色值在100以上的像素中的蓝色？

- 1) 只有A
- 2) 只有D
- 3) B和C
- 4) C和D
- 5) 一个也没有
- 6) 全部都是

其他的又做了什么呢？

- A.

```
def blueOneHundred(picture):
    for x in range(0,100):
        for y in range(0,100):
            pixel = getPixel(picture,x,y)
            setBlue(pixel,100)
```
- B.

```
def removeBlue(picture):
    for p in getPixels(picture):
        if getBlue(p) > 0:
            setBlue(p,100)
```
- C.

```
def noBlue(picture):
    blue = makeColor(0,0,100)
    for p in getPixels(picture):
        color = getColor(p)
        if distance(color,blue) > 100:
            setBlue(p,0)
```
- D.

```
def byeByeBlue(picture):
    for p in getPixels(picture):
        if getBlue(p) > 100:
            setBlue(p,0)
```

4.3 我们已经看到，如果在目标图片的坐标增加1时，源图片的坐标增加2，这样复制像素以后最终源图片在目标图片上得以缩小。如果目标图片的坐标每次也增加2会怎样呢？如果两者每次都增加0.5并用int来取整数部分又会怎样呢？

4.4 编写函数以对角线 (0, 0) ~ (width, height) 为轴制作图片镜像。

4.5 编写函数以对角线 (0, height) ~ (width, 0) 为轴制作图片镜像。

4.6 编写函数只放大图片的一部分，试着用它让某人的鼻子更长。

- 4.7 编写函数只缩小图片的一部分，用它让某人的脑袋看起来更小。
- 4.8 编写翻转图片的函数，如果一个人在往右看，把他变成往左看。
- 4.9 编写通用的裁剪函数，接受一幅源图片，起点X和Y坐标，终点X和Y坐标。创建新的图片并复制指定区域，然后返回新图片。
- 4.10 编写函数，复制图片的不同部分到目标图片的不同位置上。
- 4.11 编写通用的图片放大函数，接受一幅图片，使用makeEmptyPicture(width, height)创建并返回两倍大小的图片。
- 4.12 编写通用的图片缩小函数，接受一幅图片，使用makeEmptyPicture(width, height)创建并返回一半大小的图片。
- 4.13 使用嵌套循环修改上一章任意一个函数。检查结果，确保它仍然完成同样的功能。
- 4.14 编写函数从一幅图片中复制一块三角形区域到另一幅图片中。
- 4.15 编写函数将输入图片最左边的20列像素镜像到第20~40列。
- 4.16 编写函数，减少图片顶部1/3范围内的红色，清除底部1/3范围内的蓝色。
- 4.17 执行下面三条命令，函数分别会输出什么内容？
 - (a) testMe(1, 2, 3)
 - (b) testMe(3, 2, 1)
 - (c) testMe(5, 75, 20)

```
def testMe(p,q,r):
    if q > 50:
        print r
    value = 10
    for i in range(0,p):
        print "Hello"
        value = value - 1
    print value
    print r
```

- 4.18 编写函数makeCollage()，在空白JPEG文件7x9.5in.jpg中制作同一幅图像的拼贴图，图像至少要出现4次。（可以随意添加更多。）在4份副本中，一份可以是原图，另外3份应该是修改过的。可以缩放、裁剪或旋转图像，制作底片、变换或更改图像的颜色，暗化或亮化。

图像做好之后再制作它的镜像。可以横向或纵向（或沿其他轴线）镜像。任何方向都可以——只要保证镜像之后4幅基本图像仍然可见。

你应该用一个函数完成所有这些事情——所有的效果和组合都应该发生在单个函数makeCollage()中。当然，使用其他函数也完全没有问题，但要让程序的测试者只需调用一次setMediaPath()，将所有输入图片都放在一个mediasources目录中，然后执行makeCollage()就能看到产生、显示并返回的拼贴图。

- *4.19 考虑一下灰度算法的原理。从基础上讲，如果你知道任何可见物的亮度（比如，一幅图像、一个字母），你都可以用类似于创建拼贴图的方法用这种视觉元素（visual element）代替一个像素。试着实现这种功能。你将需要大小相同、亮度依次递增的256

种视觉元素。你可以把原始图像中的每个像素都替换为某个视觉元素，从而制作一张拼贴图。

深入学习

计算机图形学的“圣经”是《Introduction to Computer Graphics》（机械工业出版社曾引进过影印版《计算机图形学导论》。——译者注）[14]。

高级图片技术

本章学习目标

本章媒体学习目标:

- 实现可控的颜色调整, 比如去除红眼、深褐色调和色调分离。
- 使用融合法合并图像。
- 使用背景消减从背景图像中分离前景图像并理解它在什么情况下有用, 如何有用。
- 使用色键技术从背景图像中分离前景图像。
- 往已有图片中添加文本和图形。
- 使用模糊化技术平滑锯齿。

本章计算机科学学习目标:

- 使用条件式。
- 能够在向量和位图图像之间做出选择。
- 针对一项任务, 能够在编写程序和使用现有应用程序软件之间做出选择。

5.1 颜色替换: 消除红眼、深褐色调和色调分离

使用一种颜色替换另一种颜色非常简单。我们可以做全面替换, 也可以只在一个范围内替换。有了这种方法, 我们就能制作出一些有趣的整体效果, 或者通过调整效果完成某种任务, 比如把某人的牙齿变成紫色。

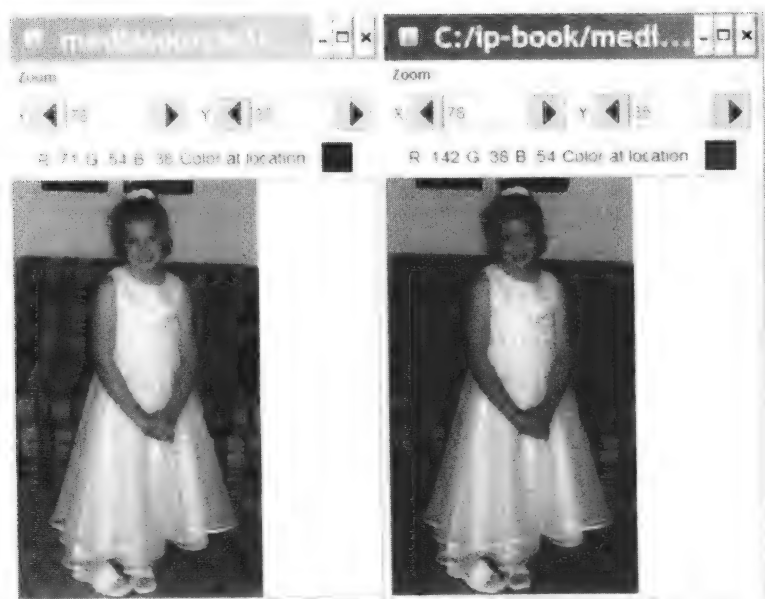


图5.1 把棕色变成红色

下面程序将Katie的头发由棕色变成红色。Mark用JES图片工具大致确定了Katie棕色头发的RGB值，然后编写程序寻找接近该值的颜色并增加这些像素的红色数量。Mark尝试了多种用于颜色间距（这里是50）和增红因子（这里是2）的值，结果Katie身后的沙发也增红了（如图5.1所示）。



程序36：把Katie变成红发女郎

```
def turnRed():
    brown = makeColor(42,25,15)
    file="C:/ip-book/mediasources/katieFancy.jpg"
    picture=makePicture(file)
    for px in getPixels(picture):
        color = getColor(px)
        if distance(color,brown)<50.0:
            redness=int(getRed(px)*2)
            blueness=getBlue(px)
            greenness=getGreen(px)
            setColor(px,makeColor(redness,blueness,greenness))
    show(picture)
    return(picture)
```

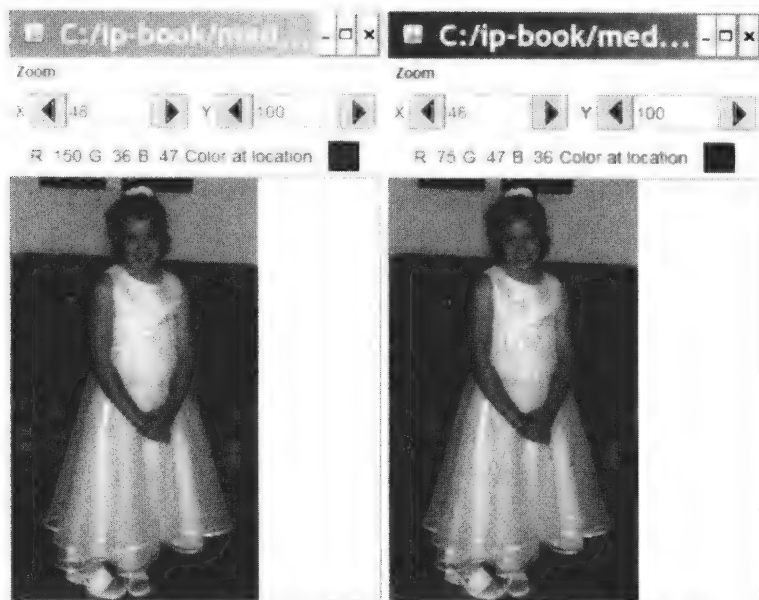


图5.2 将矩形区域内的红色加倍

程序原理

实际上，这与我们增加红色的菜谱很像，只不过使用了另外一种设置颜色的方法。

- 我们创建了颜色brown，这是我们用JES的图片工具从Katie的头发中找出来的。
- 我们创建了Katie的图片。
- 对图片中的每个像素px，我们取得它的颜色并与之前确定的颜色brown比较。我们想知道像素px的颜色是否跟brown足够接近。如何定义“足够接近”呢？我们说：如果它们的“颜色间距”（指定义在（R,G,B）颜色空间中的欧氏距离，即两种颜色各分量之差的平方和再取平方根。——译者注）在50.0以内就是足够接近。这个数字又是怎么得到的呢？我们尝试过10.0，图片几乎没变。我们也尝试过100.0，匹配的地方太多了（比如

Katie脑袋后面长沙发上的条纹也符合条件)。我们又尝试过多个不同的数字,直到获得了想要的效果。

- 如果颜色“足够接近”,我们就取出px颜色的红、绿、蓝分量,然后把红色分量乘以2,使之加倍。
- 现在,使用调整后的红色分量和原来的绿色、蓝色分量,我们把像素px置成了新的颜色。然后转向了下一个像素。

使用JES图片工具,我们也可以确定Katie脸庞附近的坐标,然后处理一下脸庞边上的棕色。最终,虽然有些效果,但没那么好。红色线太明显,而且有棱角(如图5.2所示)。



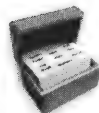
程序37: 区域内的颜色替换

```
def turnRedInRange():
    brown = makeColor(42,25,15)
    file="C:/ip-book/mediasources/katieFancy.jpg"
    picture=makePicture(file)
    for x in range(63,125):
        for y in range(6,76):
            px=getPixel(picture,x,y)
            color = getColor(px)
            if distance(color,brown)<50.0:
                redness=int(getRed(px)*2)
                blueness=getBlue(px)
                greenness=getGreen(px)
                setColor(px,makeColor(redness,blueness,greenness))
    show(picture)
    return(picture)
```

5.1.1 消除红眼

“红眼”是相机的闪光从眼底反射回来时造成的效果。实际上,消除红眼非常简单。我们找到与红色“非常接近”的像素(使用165的颜色间距效果就很好),找到后便插入替代颜色。

我们通常不会改变整幅图片。在图5.3中,Jenny穿着红色衣服。我们并不想除掉衣服上的红色,而只想改变Jenny眼睛所在的区域,从而修正红眼效果。使用JES图片工具,我们找到了眼睛的左上角和右下角坐标。这两个点是(109, 91)和(202, 107)。



程序38: 消除红眼

```
def removeRedEye(pic,startX,startY,endX,endY,
replacementcolor):
    red = makeColor(255,0,0)
    for x in range(startX,endX):
        for y in range(startY,endY):
            currentPixel = getPixel(pic,x,y)
            if (distance(red,getColor(currentPixel)) < 165):
                setColor(currentPixel,replacementcolor)
```

我们用以下方式调用函数:

```
>>> jenny = makePicture(getMediaPath("jenny-red.jpg"))
>>> explore(jenny)
>>> removeRedEye(jenny, 109, 91, 202, 107, makeColor(0,0,0))
>>> explore(jenny)
```

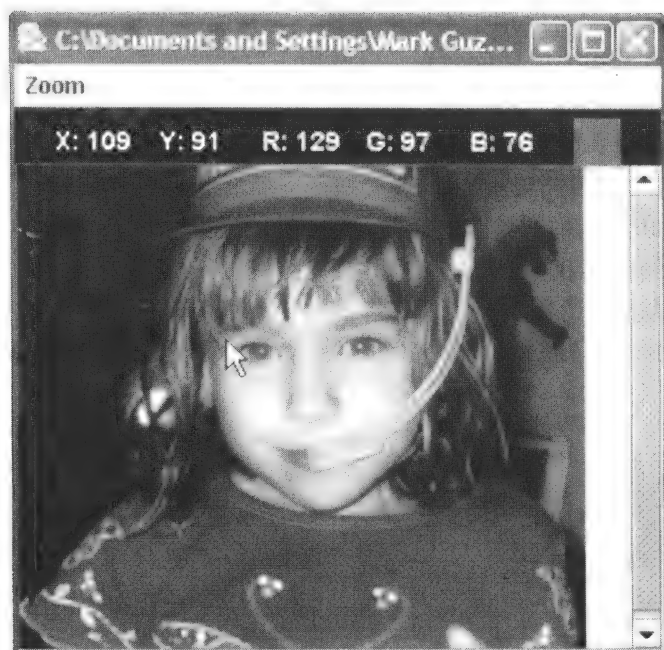


图5.3 找出Jenny的红眼睛区域 (1)

以便使用黑色取代红色——当然也可以用其他颜色来替代。效果不错，可以检验一下，现在眼睛确实只有黑色像素了（如图5.4所示）。

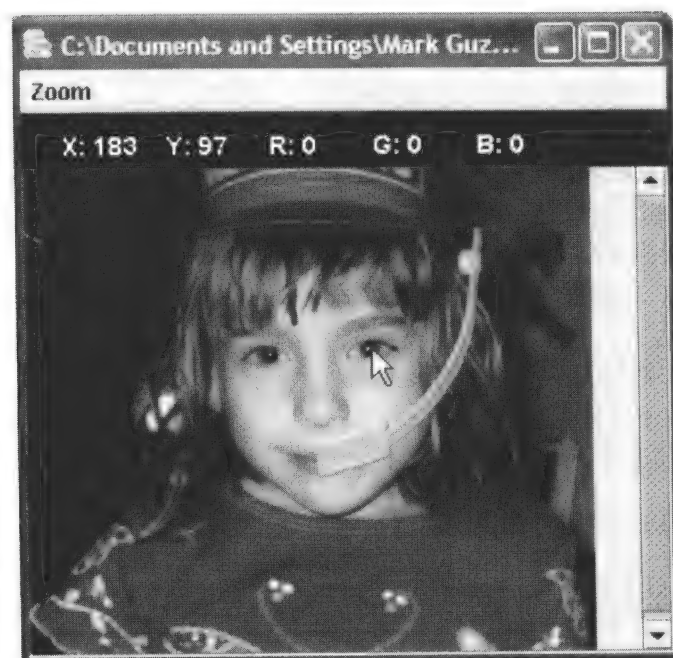


图5.4 找出Jenny的红眼睛区域 (2)

程序原理

实际上，这个算法与我们把Katie的头发变成红色的算法非常相似。

- 这个函数的写法使它可以用于不同的图片，它接受多个输入（参数），包括一幅输入图片、要改变颜色的矩形区域的坐标，以及用于取代红色的颜色，用户可以针对不同程序而改变这些输入。
- 我们把红色定义成一种深红的纯色——`makeColor(255, 0, 0)`。
- 针对输入矩形区域中的每组x和y，我们取得当前像素`currentPixel`。
- 我们检查`currentPixel`与红色是否足够接近，方法是在一个阈值（threshold value）范围内检查颜色间距。我们尝试了不同的颜色间距。最终，针对能引人注意的多数红眼，我们把颜色间距设定为165。
- 然后，我们用`replacementColor`取代了像素`currentPixel`处“足够接近”的颜色。



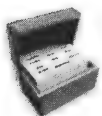
图5.5 原来的风景（左）和使用深褐色调菜谱转换过的（右）

5.1.2 深褐色调和色调分离：使用条件式选择颜色

到目前为止，我们都是通过用一种颜色替代另一种颜色的简单方法来做颜色消减。我们还可以做一些更高级的交换颜色的动作。可以用if寻找一块颜色区域，然后替换原始颜色的某些效果，或者把它改成特定颜色。结果会相当有趣。

举个例子，假如我们想制作深褐色调（sepia-toned）的印刷品。老一点的印刷品有时会有一点泛黄的色泽。我们可以直接做整体的颜色调整，但最终结果从美术角度来看不尽如人意。通过寻找不同的颜色（高亮的、阴暗的），然后以不同的方式处理它们，我们可以得到更好的效果（如图5.5所示）。

采用的方法是首先把所有的颜色都转换成灰度，这一方面是因为老的印刷品都是灰度图，另一方面也因为这会使图像更容易处理。然后我们寻找颜色亮度的高、中、低区域并分别调整它们。（这些具体的值是怎么来的呢？反复试验！不断调整直到我们对效果满意。）



程序39：将图片转换成深褐色调

```
def sepiaTint(picture):
    # 将图像转换成灰度图
    grayScaleNew(picture)

    # 遍历图片，为像素调色
    for p in getPixels(picture):
        red = getRed(p)
        blue = getBlue(p)

    # 调节低亮像素
```



```

if (red < 63):
    red = red*1.1
    blue = blue*0.9
# 调节中亮像素
if (red > 62 and red < 192):
    red = red*1.15
    blue = blue*0.85
# 调节高亮像素
if (red > 191):
    red = red*1.08
    if (red > 255):
        red = 255
    blue = blue*0.93
# 设置新的颜色值
setBlue(p, blue)
setRed(p, red)

```

程序原理

函数首先接受一幅图片输入，然后使用`grayScaleNew`函数将它转换为灰度图。（我们建议将`grayScaleNew`函数复制到程序区中与`sepiaTint`放在一起——只是这里没有显示。）对每一个像素，我们取得它的红色和蓝色。我们知道红色和蓝色值是一样的，因为它现在是灰度图，但现在我们是要修改红色和蓝色。我们寻找特定的颜色范围然后用不同的方式处理它们。

注意，在高亮部分（光度最亮的区域）调色时，我们在`if`块内部还有一个`if`。这里我们是不想让颜色值回绕——如果红色值太高，我们想让它255封顶。最后，我们把红色和蓝色值设成了新的红色和蓝色值并转向下一个像素。

色调分离（posterizing）是一种非常类似的过程，其结果是将图片转换成含有更少的颜色数目。实现方法是寻找特定的颜色范围，然后把该范围内的颜色改成同样的值。结果就是图片中的颜色数目减少了（如图5.6所示）。比如，在下面的菜谱中，如果红色值是1、2、3…或64，我们就把它变成31。这样，我们消除了一块完整的红色范围，并把它全部置成了一种特定的红色值。我们在一名计算机系学生Anthony的照片上尝试了这一过程。

```

>>> file = "c:/ip-book/mediasources/anthony.jpg"
>>> student = makePicture(file)
>>> explore(student)
>>> posterize(student)
>>> explore(student)

```



程序40：图片的色调分离

```

def posterize(picture):
    # 循环遍历像素
    for p in getPixels(picture):
        # 获得RGB值
        red = getRed(p)
        green = getGreen(p)
        blue = getBlue(p)

```

```

# 检查并设置红色值
if(red < 64):
    setRed(p, 31)
if(red > 63 and red < 128):
    setRed(p, 95)
if(red > 127 and red < 192):
    setRed(p, 159)
if(red > 191 and red < 256):
    setRed(p, 223)

# 检查并设置绿色值
if(green < 64):
    setGreen(p, 31)
if(green > 63 and green < 128):
    setGreen(p, 95)
if(green > 127 and green < 192):
    setGreen(p, 159)
if(green > 191 and green < 256):
    setGreen(p, 223)

# 检查并设置蓝色值
if(blue < 64):
    setBlue(p, 31)
if(blue > 63 and blue < 128):
    setBlue(p, 95)
if(blue > 127 and blue < 192):
    setBlue(p, 159)
if(blue > 191 and blue < 256):
    setBlue(p, 223)

```

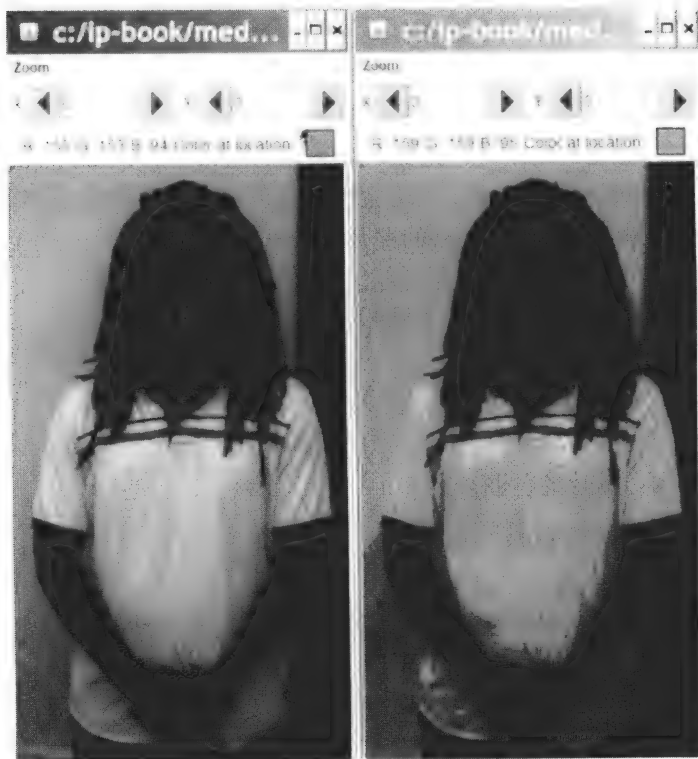


图5.6 把原始图片（左）中的颜色数目减少（右）

一种有趣的效果来自于灰度和色调的同时使用。实现方法是计算一个亮度值，然后只把像素的颜色设成两种：要么黑色要么白色——只有两级。结果是一幅看起来像图章或炭画一样的图片（如图5.7所示）。



程序41：色调分离成两级灰度

```
def grayPosterize(pic):
    for p in getPixels(pic):
        r = getRed(p)
        g = getGreen(p)
        b = getBlue(p)
        luminance = (r+g+b)/3
        if luminance < 64:
            setColor(p,black)
        if luminance >= 64:
            setColor(p,white)
```



图5.7 图片的两级灰度色调分离

5.2 合并像素：图片模糊化

图片放大后常会导致粗糙的边缘：线条上出现小“阶梯”，我们称之为**像素化**（pixelation）（参阅<http://en.wikipedia.org/wiki/Pixelation>。——译者注）。可以通过图像模糊（blurring）技术来减轻像素化：有目的地让某些尖利的边缘“软化”（即更加平滑，呈曲线型）。

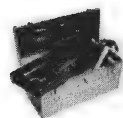
模糊化的方法（算法）有很多。这里我们使用非常简单的一种，把各个像素设成它周围像素的平均值。



程序42：简单的模糊化

```
def blur(filename):
    source=makePicture(filename)
    target=makePicture(filename)
    for x in range(1, getWidth(source)-2):
        for y in range(1, getHeight(source)-2):
            top = getPixel(source,x,y-1)
            left = getPixel(source,x-1,y)
            bottom = getPixel(source,x,y+1)
            right = getPixel(source,x+1,y)
            center = getPixel(target,x,y)
            newRed=(getRed(top)+ getRed(left) + getRed(bottom) + getRed(right)+
+ getRed(center))/5
            newGreen=(getGreen(top) + getGreen(left) + getGreen(bottom)+
+ getGreen(right)+getGreen(center))/5
            newBlue=(getBlue(top) + getBlue(left) + getBlue(bottom) + getBlue(
+ (right)+ getBlue(center))/5
            setColor(center, makeColor(newRed, newGreen, newBlue))
    return target
```

-这几行程序应该与下面的一行连续。Python中一条命令不可以跨多行。



实践技巧：Python中不要在命令中间断行

在上面这样的例子中，你会看到“自动换行”。应该位于同一行的语句实际显示在两行上。为了让代码适合书页尺寸只能这样，但我们不能在Python中真的这样断行。在语句或表达式结束之前，我们不能敲回车来断行。

图5.8显示了拼贴图中的花朵，先放大，然后又做了模糊处理。你可以从放大的版本中看到像素化——明显的块状边缘。模糊化之后，某些地方的像素化去掉了。更细致的模糊化会考虑颜色区域（因此不同颜色之间的边缘仍然明显），从而在减轻像素化的时候不会影响明显的颜色分界。

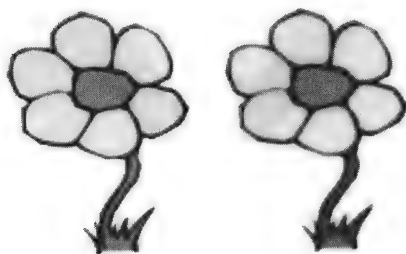


图5.8 将花朵放大（左），然后模糊化以减轻像素化（右）

程序原理

我们通过 `newPic = blur(pickAFile())` 这样的命令来使用这个函数。然后，我们基于输入的文件名创建了两份拷贝。我们只修改 `target`，这样就可以一直基于 `source` 中原先的颜色求平均值。我们走遍了从1到宽度减1的 x 坐标——这里，我们显式利用了 `range` 不包括终点值的事实。对 y 坐标也做同样处理。这样做的理由是我们将对每个 x 和 y 坐标做加1和减1操作以求得平均值。如此一来，我们就不会平均边上的像素，否则判断起来会有点麻烦。对每个 (x, y) ，我们取得它左边的像素 $(x-1, y)$ ，右边的像素 $(x+1, y)$ ，上面的像素 $(x, y-1)$ 和下面的像素 $(x, y+1)$ ，以及位于中心的像素本身 (x, y) （它直接来自 `target`，因为我们确定此时 (x, y) 还没有改变）。我们分别计算5个像素的红色、绿色和蓝色平均值，然后把 (x, y) 点的颜色设成平均颜色。

效果非常好（如图5.8所示），但我们完全可以做得更好。像这样简单的模糊化会丢失一些细节。如果我们在使用模糊化消减像素化的同时能保持原来的细节，结果会怎样呢？我们又如何做到呢？如果我们在计算平均值之前检查亮度值——或许我们不该跨明显的亮度边界来模糊化，因为那样会消减一些细节——结果又会怎样呢？这只是想法之一——关于图像模糊化，好的算法很多。

5.3 比较像素：边缘检测

模糊化是一种基于多个像素计算平均值的过程。边缘检测（edge detection）是一种类似的过程，在这一过程中，我们通过比较像素来决定为某个像素设置什么颜色，但基本上我们只会将像素设成黑色或白色。其思想是：尝试像艺术家速写那样画线条。

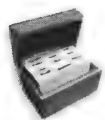
我们可以从人物的线条画中识别出人脸或其他特征，这真是视觉系统的一项了不起的特性。看看周围的世界，并没有什么明显的线条定义世界的特征。鼻子和眼睛周围没有明晰的线条，但任何一个小孩都能画出一张脸，上面有一个钩代表鼻子，两个圈代表眼睛——而且我们都能认出那是一张脸！通常，在有亮度差别的地方，我们就能看到一条线。



图5.9 转换成“线条画”（右）的蝴蝶（左）

我们可以尝试为这一过程建模（如图5.9所示）。在下面的菜谱中，我们把每个像素的亮度

与它下面和右边的像素进行比较。如果下面和右边都有适当的亮度差，那么我们就把像素置成黑色（实验发现用10做阈值效果非常不错），否则，把它置为白色。



程序43：使用边缘检测创建简单的线条画

```
def lineDetect(filename):
    orig = makePicture(filename)
    makeBw = makePicture(filename)
    for x in range(0,getWidth(orig)-1):
        for y in range(0,getHeight(orig)-1):
            here=getPixel(makeBw,x,y)
            down=getPixel(orig,x,y+1)
            right=getPixel(orig,x+1,y)
            hereL=(getRed(here)+getGreen(here)+getBlue(here))/3
            downL=(getRed(down)+getGreen(down)+getBlue(down))/3
            rightL=(getRed(right)+getGreen(right)+getBlue(right))/3
            if abs(hereL-downL)>10 and abs(hereL-rightL)>10:
                setColor(here,black)
            if abs(hereL-downL)<=10 and abs(hereL-rightL)<=10:
                setColor(here,white)
    show(makeBw)
    return makeBw
```

程序原理

与灰度色调分离类似，目标也是将每个像素设成黑色或白色，设置的依据是看是否有亮度差别。

- 我们使用 `bwVersion = lineDetect(pickAFile())` 这样的命令调用函数。基于输入的文件名，我们创建了两份图片拷贝——源图片 `orig` 和目标图片 `makeBw`。
- 我们使用从1到宽度减1的下标 `x` 和从1到高度减1的下标 `y`，这里我们依赖 `range` 不包括最后一个值的事实。我们将把 (x, y) 点的像素同 $(x + 1, y)$ （右侧）点的像素和 $(x, y + 1)$ （下方）点的像素比较。
- 我们取得当前点的像素 `here`，它下方的像素 `down` 和它右侧的像素 `right`。
- 我们计算这三个像素的亮度。
- 用待比较像素的亮度（`hereL`）分别减去右侧像素的亮度（`rightL`）和下方像素的亮度（`downL`），如果差的绝对值（`abs`）大于阈值10，那么我们就把像素置成黑色。使用绝对值是因为我们只关心两个亮度值的差异。我们并不关心究竟哪个值更大。
- 如果亮度的差没有大于阈值（小于或等于它），那么我们就把像素置为白色。

边缘检测和线条绘制还有更好的算法。举例来说，这里我们只是将各个像素设成黑色或白色，没有真正考虑“线”的概念。我们可以使用图片模糊这样的技术来平滑图像，使各个点连起来更像一条线。我们也可以在修改像素时考虑它附近的像素，仅在附近像素也要置成黑色时才把它置黑，也就是绘制一条线而不只是画点。

5.4 图片融合

本章讨论基于其他图像来创建图片的技术。我们将使用新的方法制作图片（比如，把某人从背景中取出，然后放到新的环境中），或者从头开始制作图片而不是显式地设置各个像素。

基于图片合并来创建新图片的方法之一是混合像素的颜色以反映两幅图片。通过复制来创建拼贴图的时候，任何重叠部分通常都意味着一张图片会显示在另一张之上。最后绘出的图片

就是显示在另一张之上的图片。不一定非要这样，我们可以把图片的颜色叠加起来，从而实现图片的融合（blend）。这样的融合可以提供透明效果。

我们知道，100%是一个整体，两样东西各取50%也可以凑成一个整体。在下面的菜谱中，我们通过70列像素（Barbara相片的宽度减150）的叠加融合了母亲和女儿的照片（如图5.10所示）。



程序44：融合两幅图片

```
def blendPictures():
    barb = makePicture(getMediaPath("barbara.jpg"))
    katie = makePicture(getMediaPath("Katie-smaller.jpg"))
    canvas = makePicture(getMediaPath("640x480.jpg"))

    # 复制Barb前面的150列像素
    sourceX=0
    for targetX in range(0,150):
        sourceY=0
        for targetY in range(0,getHeight(barb)):
            color = getColor(getPixel(barb,sourceX,sourceY))
            setColor(getPixel(canvas,targetX,targetY),color)
            sourceY = sourceY + 1
        sourceX = sourceX + 1
    # 现在取出Barb剩下的部分
    # Barb和Katie各占50%
    overlap = getWidth(barb)-150
    sourceX=0
    for targetX in range(150,getWidth(barb)):
        sourceY=0
        for targetY in range(0,getHeight(katie)):
            bPixel = getPixel(barb,sourceX+150,sourceY)
            kPixel = getPixel(katie,sourceX,sourceY)
            newRed= 0.50*getRed(bPixel)+0.50*getRed(kPixel)
            newGreen=0.50*getGreen(bPixel)+0.50*getGreen(kPixel)
            newBlue = 0.50*getBlue(bPixel)+0.50*getBlue(kPixel)
            color = makeColor(newRed,newGreen,newBlue)
            setColor(getPixel(canvas,targetX,targetY),color)
            sourceY = sourceY + 1
        sourceX = sourceX + 1
    # Katie剩下的像素列
    sourceX=overlap
    for targetX in range(150+overlap,150+getWidth(katie)):
        sourceY=0
        for targetY in range(0,getHeight(katie)):
            color = getColor(getPixel(katie,sourceX,sourceY))
            setColor(getPixel(canvas,targetX,targetY),color)
            sourceY = sourceY + 1
        sourceX = sourceX + 1
    show(canvas)
    return canvas
```

程序原理

这个函数分为三个部分——有Barb没有Katie的部分、两者都有的部分和只有Katie的部分。

- 我们首先为barb、katie和目标canvas创建图片对象。
- 前面的150列，我们只把barb的像素复制到画布中。
- 接下来是实际融合的部分。targetX从150开始，因为画布中已经有来自barb的150列，我们使用sourceX和sourceY同时索引barb和katie，但由于我们已经复制了barb的150

列像素，所以索引barb时我们必须把sourceX加上150。y坐标只到达katie照片的高度，因为它比barb的照片短一些。

- 融合发生在循环体中。我们从barb中取一个像素命名为bPixel，再从katie中取一个像素命名为kPixel。然后，我们为目标像素（位于targetX和targetY处的像素）计算红、绿、蓝分量，方法是从两个源像素的红、绿、蓝中各取50%。
- 最后，我们复制katie剩下的像素并结束。



图5.10 融合妈妈和女儿的照片

5.5 背景消减

假如你有一张三人的照片，还有一张他拍照时站的地方但上面没有这个人的照片（如图5.11所示）。你能把这个人的背景除去（即算出颜色完全相同的部分），然后换上另一个背景吗？比如月球（如图5.12所示）？



程序45：消除背景并换上新的背景

```
def swapBack(pic1, back, newBg):
    for x in range(0,getWidth(pic1)):
        for y in range(0,getHeight(pic1)):
            p1Pixel = getPixel(pic1,x,y)
            backPixel = getPixel(back,x,y)
            if (distance(getColor(p1Pixel),
                        getColor(backPixel)) < 15.0):
                setColor(p1Pixel,getColor(getPixel(newBg,x,y)))
    return pic1
```

程序原理

函数swapBack（swap background的简写）接受一幅图片（上面同时有前景和背景），一幅背景图片和一幅新背景。针对输入图片中的所有像素，我们做以下操作：

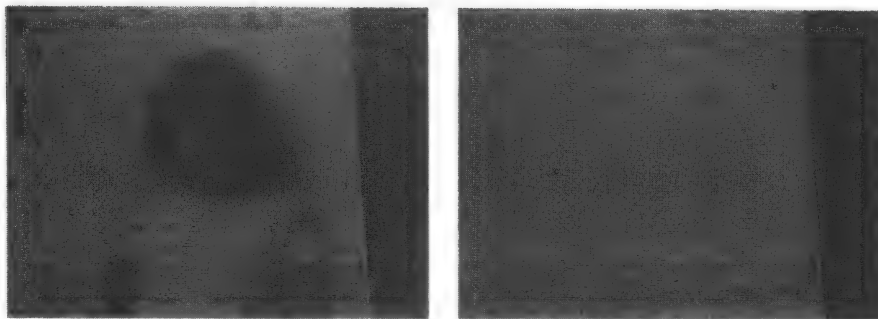


图5.11 一张小孩（Katie）的照片和上面没有她的背景照片



图5.12 新的背景：月球

- 同时从原图片和背景图片中取得（相同坐标下的）匹配像素。
- 计算颜色间距。这次我们使用阈值15.0，但你可以尝试一下其他值。
- 如果间距较小（小于阈值），那么我们便假定这个像素是背景的一部分。然后，我们从新背景中取得同一坐标处的颜色，并将输入图片中的像素设成这种颜色。

你可以试一试这个程序，结果不是很好（如图5.13所示）。我女儿穿的T恤颜色与墙的颜色太接近了，导致月球都融到T恤中去了。虽然光线较暗，但图片上的阴影还是对结果有影响。背景图片中没有阴影，于是算法将阴影部分作为前景看待了。这种结果表明：背景和前景的色差对于背景消减非常重要——因此好的灯光很重要！



图5.13 Katie在月球上

Mark使用一张两个学生站在花砖墙之前的照片做了同样的事情。拍照时他使用了三脚架(这对像素的对齐非常重要),却不幸忘了关闭自动聚焦功能,因此两幅原图(如图5.14所示)不完全适配。使用丛林风景做的背景交换几乎没什么效果。Mark把阈值改成50,最终背景被交换了一部分(如图5.15所示)。

仅靠改变阈值不一定能改善效果。它显然能让更多前景识别为背景。然而,对于颜色匹配导致的背景渗透到衣服中这种问题,改变阈值不会有很大帮助。



图5.14 两个人站在墙壁前面(左)和墙壁的图片(右)



图5.15 使用背景消减技术用丛林交换墙壁,阈值为50

5.6 色键

电视上的天气预报员会在一张地图前面用手示意风暴前锋的到来。实际上,录像时他们站在固定颜色(通常是蓝色或绿色)的背景前面,之后再通过数字技术用所需地图中的像素来替换背景颜色,这叫色键(chromakey)技术。替换一种已知的颜色相对容易一些,也不会对灯光那么敏感。Mark拿了我儿子的蓝色床单,在家庭娱乐中心悬挂起来,然后坐在它前面利用相机的定时器功能为自己拍了一张照片(如图5.16所示)。



程序46：色键：使用新的背景替换蓝色

```
def chromakey(source,bg):
    # 源图片中应该有东西位于蓝色之前，bg是新背景
    for x in range(0,getWidth(source)):
        for y in range(0,getHeight(source)):
            p = getPixel(source,x,y)
            # 我对蓝色的定义：红 + 绿 < 蓝
            if (getRed(p) + getGreen(p) < getBlue(p)):
                # 然后从新的背景中取出同一位置的颜色
                setColor(p,getColor(getPixel(bg,x,y)))
    return source
```



图5.16 Mark坐在蓝色床单前

程序原理

这一次，我们只接受一幅源图片（图片上同时有前景和背景）和新的背景bg。它们必须大小相同！Mark使用JES图片工具为这份菜谱找出了一种识别蓝色的规则。他不想使用相等来判断，甚至不想使用到颜色（0，0，255）的间距，因为它知道很少有蓝色正好是完全的蓝色。他发现，那些可认为是蓝色的像素往往红色和绿色值都很小，常常蓝色值大于红、绿两色之和。于是这成了他在菜谱中寻找像素的规则。只要一个像素的蓝色值大于红、绿两色的和，那么，他就把新背景中对应像素的颜色交换进来。

效果真的很棒（如图5.17所示）。但还是要注意一点，月球表面有了“褶皱”。真正酷的地方在于这份菜谱适用于任何一张与源图像大小相同的背景图（如图5.18所示）。

这段代码还可以换种写法，它更短，但完成的功能一样。这种写法使用getX和getY函数来求出坐标，但这导致setColor语句显得凌乱了一些。



程序47：更短的色键程序

```
def chromakey2(source,bg):
    for p in pixels(source):
        if (getRed(p)+getGreen(p) < getBlue(p)):
            setColor(p,getColor(getPixel(bg,getX(p),getY(p))))
    return source
```

没有人会用常见颜色做色键，比如红色——人的脸上就有许多红色。Mark用图5.19中的两张图片试验了一下——拍的时候其中的一张打开了闪光灯，另一张没有。他把检查颜色的条件换成了`if getRed(p) > (getGreen(p) + getBlue(p))`。结果没用闪光灯的那张很吓人，学生的脸变得“丛林化”了。用了闪光灯的一张好一些，但交换之后闪光依然可见（如图5.20所示）。为什么电影和电视天气预报的制作人都使用蓝色或绿色做色键呢？原因很明显——这两种颜色与常见颜色，比如人脸上的颜色重复的机会更小。



图5.17 月球上的Mark

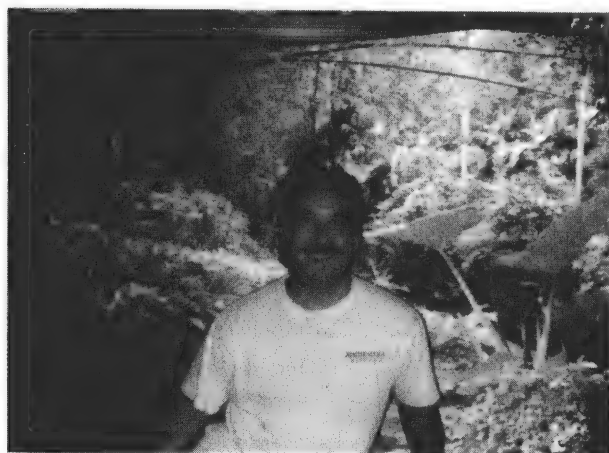


图5.18 丛林中的Mark

实现色键的专业设备和软件采用的过程与这里的稍有不同。我们的算法首先查找需要替换的颜色，然后进行替换。专业的色键处理过程会产生一个掩码（Mask）位图。掩码位图与原始图像大小相同，那些需要改变的像素在掩码中对应的点是白色的，不应改变的像素对应的点是黑色的。然后，使用掩码位图来决定哪些像素需要改变。使用掩码的一个优势在于分离了（a）检测哪些像素需要改变；与（b）改变像素这两个过程。把它们分开，更便于单独改善其中的一个，从而改善整体效果。



图5.19 红色背景前的学生，不用闪光灯（左）和使用闪光灯（右）

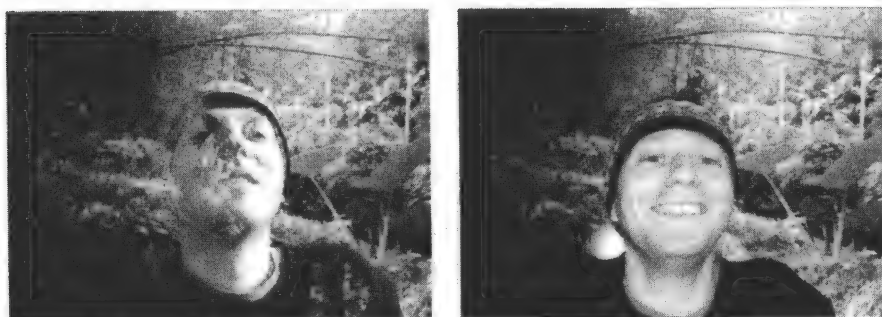


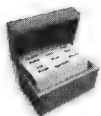
图5.20 使用色键菜谱处理红色背景，不用闪光灯（左）和使用闪光灯（右）

5.7 在图像上绘图

有时，你会考虑从零开始创建属于自己的图像。我们知道这并不困难，只需要把像素颜色设置成想要的值，但通过设置单个像素来画线、画圆或者画一些字母还是有点难度的。在图像上绘图，方法之一就是直接根据图形设置相应的像素。下面就是一个例子，在佐治亚理工学院计算机系一名学生Carolina的照片上画横线和竖线（如图5.21所示）。程序要求你选取一个文件，它基于文件创建一幅图片。然后，程序调用`verticalLines`函数在图片上每隔5个像素画一道竖线。之后它又调用`horizontalLines`函数每隔5个像素画一条横线。最后，它显示并返回了结果图片。



图5.21 原来的（左）和加过线条的（右）Carolina



程序48：通过设置像素来画线

```
def lineExample():
    img = makePicture(pickAFile())
    verticalLines(img)
    horizontalLines(img)
    show(img)
    return img

def horizontalLines(src):
    for x in range(0,getHeight(src),5):
        for y in range(0,getWidth(src)):
            setColor(getPixel(src,y,x),black)

def verticalLines(src):
    for x in range(0,getWidth(src),5):
        for y in range(0,getHeight(src)):
            setColor(getPixel(src,x,y),black)
```

注意：这个程序使用了颜色名字black。你也许还记得JES为我们预定义了很多颜色：black、white、blue、red、green、gray、lightGray、darkGray、yellow、orange、pink、magenta和cyan。你可以任意使用这些名字，就像使用其他函数和命令一样。

可以想象，只要像这样把单个像素设成想要的颜色，我们就能画出任何想画的东西。只要计算出哪些像素需要变成哪种颜色，我们就可以画矩形和圆形。我们甚至可以画字母——把适当的像素设成适当的颜色，我们可以画出任何一个字母。虽然这是可行的，但需要针对各种不同形状和字母做大量的数学计算。这种计算是许多人都需要的，因此人们已经把基本绘图工具做成了库。

5.7.1 使用绘图命令

大多具有图形库的现代编程语言都提供了一组函数，使我们能直接在图片上绘制各种不同的形状和文本。下面列出了一些这样的函数：

- addText(pict, x, y, string)将字符串输出到图片中从(x, y)开始的地方。
- addLine(pict, x1, y1, x2, y2)从(x1, y1)到(x2, y2)画一条直线。
- addRect(pict, x1, y1, w, h)使用黑色线条画矩形，矩形的左上角在(x1, y1)，宽度为w，高度为h。
- addRectFilled(pict, x1, y1, w, h, color)画一个矩形，使用输入的颜色color填充，矩形的左上角在(x1, y1)，宽度为w，高度为h。

我们可以使用这些命令在已有的图片上添加东西。如果沙滩上漂过来一个神秘的红色盒子，会是什么效果呢？我们可以用如下的命令来显现这一景象（如图5.22所示）。



程序49：在沙滩上添加一个盒子

```
def addABox():
    beach = makePicture(getMediaPath("beach-smaller.jpg"))
    addRectFilled(beach,150,150,50,50,red)
    show(beach)
    return beach
```




图5.22 沙滩上漂过来一个盒子

下面是使用这些绘图命令的另外一个例子（如图5.23所示）。



程序50：使用绘图命令的一个例子

```
def littlepicture():
    canvas=makePicture(getMediaPath("640x480.jpg"))
    addText(canvas,10,50,"This is not a picture")
    addLine(canvas,10,20,300,50)
    addRectFilled(canvas,0,200,300,500,yellow)
    addRect(canvas,10,210,290,490)
    return canvas
```

This is not a picture

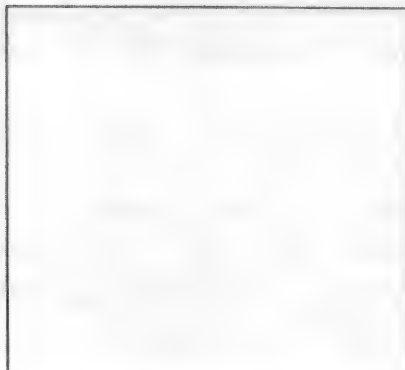


图5.23 画出来的一幅小图片

5.7.2 向量和位图表示

考虑这样一个问题，图片（如图5.23所示）和菜谱哪一个更小？图5.23存到磁盘上时大约占15 KB。菜谱50 B还不到100 B。但对许多应用场景来说，它们都是等同的。如果只保存程序，而没有保存像素，那么结果会怎样？图形的**向量表示**（vector representation）就与这个问题有关。

基于向量的图形表示是在需要时可以产生图片的可执行程序。这种表示被用于PostScript、Flash和AutoCAD中。当你在Flash或AutoCAD中改变图像时，实际改变的是其底层表示——本质上讲，你是在改变程序50那样的程序。这时程序再次执行并将图像显示出来。然而，感谢摩尔定律，执行和重新显示发生得如此之快，以至于感觉上你就是在改变图片本身。

像PostScript和TrueType这样的字体定义语言，实际上就是为每一个字母或符号定义了一个微型的程序（或公式）。需要一个特定大小的字母或符号时，程序就会运行，计算出哪些像素需要设置成哪些值。（有些字体格式指定两种以上的颜色来产生平滑的曲线效果。）由于程序接受指定字体尺寸的输入参数，字母和符号可以基于任何尺寸来产生。

另一方面，位图图形表示保存每一个像素或像素的压缩表示。像BMP、GIF和JPEG这样的格式本质上都是位图表示。GIF和JPEG是压缩表示——它们并没有使用24位来表示每一个像素。相反，它们使用更少的位来表示同样的信息。

压缩（compression）是什么意思呢？它是指用来让文件更小的各种技术。有些压缩技术是**有损压缩**（lossy compression）——有些细节会丢失，但可以指望丢失的都是最不重要的细节（甚至可能是人的眼睛和耳朵感知不到的）。另一些称为**无损压缩**（lossless compression）的技术不丢失任何细节，但仍然能使文件缩紧。无损压缩技术的例子之一是**行程长度编码**（Run Length Encoding, RLE）。

想象某幅图片中有一长串黄色像素，旁边被蓝色像素围着。就像：

B B Y Y Y Y Y Y Y Y Y B B

如果不把这些内容编码成一长串像素，而是像下面这样，会怎样呢？

B B 9 Y B B

用文字来说，你编码的是“蓝、蓝，然后9个黄，然后蓝和蓝”。由于每个黄色像素占24位（3字节分别表示红、绿、蓝），但记录“9”只需要一个字节，结果就节省了大量空间。我们说：我们编码了黄色行程的长度——也就是编码行程长度。这只是用来使图片更小的压缩方法之一。

基于向量的表示在许多方面都优于位图表示。如果你能用基于向量的格式来表示一幅想要发送出去的图片（比如通过因特网），那比发送所有像素要节省得多——从某种意义上讲，向量表示本身就是压缩过的。本质上，你发送的是绘制图片的指令，而不是图片本身。然而，对于非常复杂的图像，指令会与图像一样长（设想一下把绘制蒙娜·丽莎的指令发送出去），这种情况下，向量表示也就不再有优势。但当图像足够简单的时候，比起同样的JPEG图像，像Flash中使用的那种表示的上传下载时间都会更快。

基于向量表示法的真正优势出现在你想改变图片的时候。假如你正在绘制一张建筑图纸，想利用绘图工具来延长一条直线。如果你的绘图工具只能处理位图图像（有时称为**图画工具**），那么你所拥有的只是屏幕上更多的像素，它们靠近屏幕上另外一些表示直线的像素。计算机中没有任何信息表明哪些像素表示哪一种线条——它们仅仅是像素。但如果你的绘图工具能处理

基于向量的表示（有时称为图形工具），那么延长一条直线就意味着改变底层的直线表示。

这有什么重要的呢？底层的表示实际是图画的规格说明，它可以用于任何需要规格说明的地方。设想一下，拿来一幅零件的图纸，然后基于这张图纸在切割机和冲压机上实际操作。这是许多工场里每天都在发生的事情，而之所以能够这样，是因为图纸不只是像素——它是线条和线条之间关系的规格说明，可以缩放，可以用来决定机器的行为。

你可能疑惑：“我们又如何修改程序呢？我们可以编写一个程序，它的功能从本质上讲是重新输入另一个程序或者程序的一部分吗？”是的，我们可以。第三部分将对这个问题进行介绍。

5.8 指定绘图过程的程序

这样的绘图函数可用来创建具有精确规格的图片——这是一类很难手工完成的事情。举个例子，看图5.24。

这是自动生成一种著名的视觉错觉的过程，但它不像那些著名的错觉图片那样效果明显——但这个版本的原理更易于理解。眼睛会告诉我们图片的左半边比右半边更亮，尽管分别位于两端的1/4灰度是完全一样的。出现这种效果是由于中间部分明显的边界，一边（从左往右）从灰逐渐变白，另一边从黑逐渐变灰。

图5.24中的图像是经过细致定义产生出来的。这样的图片很难用铅笔在纸上画出来。使用像Photoshop那样的软件倒是可以，但也不会太容易。然而，使用本章介绍的图形函数，我们可以方便而精确地指定那幅图片应该是怎样的。

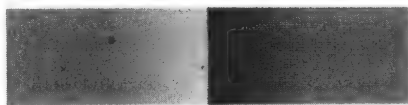


图5.24 程序实现的灰度效果



程序51：绘制灰度效果

```
def grayEffect():
    file = getMediaPath("640x480.jpg")
    pic = makePicture(file)
    # 第一部分，100列灰度级为100的像素
    gray = makeColor(100,100,100)
    for x in range(0,100):
        for y in range(0,100):
            setColor(getPixel(pic,x,y),gray)
    # 第二部分，100列灰度级逐渐增加的像素
    grayLevel = 100
    for x in range(100,200):
        gray = makeColor(grayLevel, grayLevel, grayLevel)
        for y in range(0,100):
            setColor(getPixel(pic,x,y),gray)
        grayLevel = grayLevel + 1
    # 第三部分，100列灰度级从0开始逐渐增加的像素
    grayLevel = 0
    for x in range(200,300):
        gray = makeColor(grayLevel, grayLevel, grayLevel)
        for y in range(0,100):
            setColor(getPixel(pic,x,y),gray)
```

```

        grayLevel = grayLevel + 1
    # 最后一部分, 100列灰度级为100的像素
    gray = makeColor(100,100,100)
    for x in range(300,400):
        for y in range(0,100):
            setColor(getPixel(pic,x,y),gray)
    return pic

```

图形函数最适合重复绘制那种线条和图形的位置以及颜色的选取都能用数学关系来确定的图形。



程序52: 绘制图5.25中的图片

```

def coolPic():
    canvas=makePicture(getMediaPath("640x480.jpg"))
    for index in range(25,0,-1):
        color = makeColor(index*10,index*5,index)
        addRectFilled(canvas,0,0,index*10,index*10,color)
    show(canvas)
    return canvas

```

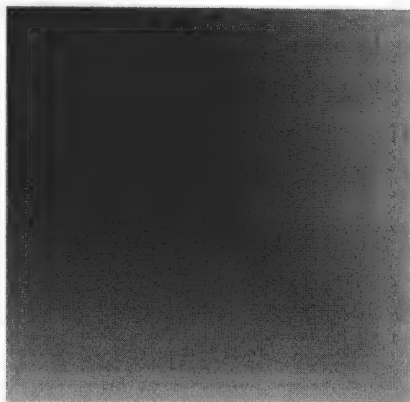


图5.25 嵌套的彩色矩形图像



程序53: 绘制图5.26中的图片

```

def coolPic2():
    canvas=makePicture(getMediaPath("640x480.jpg"))
    for index in range(25,0,-1):
        addRect(canvas,index,index,index*3,index*4)
        addRect(canvas,100+index*4,100+index*3,index*8,index*10)
    show(canvas)
    return canvas

```

我们为什么编写程序

我们为什么要编写程序, 特别是绘制图片的程序? 我们可以在Photoshop或Visio中画这些图片吗? 当然可以, 但我们必须懂得怎么做, 而这样的知识并不容易获得。我们能通过Photoshop教会你这些知识吗? 可能吧, 但那样很费劲——Photoshop不简单。

但如果我们把这些程序给你, 你可以随时创建图片。而且, 给了你这些程序, 就等于给了

你一些可根据自己的需要而调整的精确定义。



图5.26 嵌套的空白矩形图像



计算机科学思想：我们编写程序来封装并传达过程

我们编写程序是为了精确地描述一种过程并把它传达给其他人。

设想你有一些想要传达给别人的过程。不一定是画图——可以设想它是一种财务处理过程（可以用电子表格或用Quicken那样的程序来完成）或者是对文本所做的某些处理（比如编排一本书或宣传册）。如果能手工完成，那就应该手工去做。如果你需要教另外一个人如何完成它，那么可以考虑编写一个程序来做。如果你需要向许多人解释如何完成它，那肯定要使用程序了。如果你想让许多人都能够自己完成这一过程，而不需要某个人先教会他们一些东西，那就必然要编写程序并把它提供给人们。

编程摘要

以下是本章介绍的函数：

<code>addText(pict, x, y, string)</code>	将字符串输出到图片中从 (x, y) 开始的地方
<code>addLine(pict, x1, y1, x2, y2)</code>	从 $(x1, y1)$ 到 $(x2, y2)$ 画一条直线
<code>addRect(pict, x1, y1, w, h)</code>	使用黑线画矩形，矩形的左上角在 $(x1, y1)$ ，宽度为 w ，高度为 h
<code>addRectFilled(pict, x1, y1, w, h, color)</code>	画一个矩形，使用输入的颜色 <code>color</code> 填充，矩形的左上角在 $(x1, y1)$ ，宽度为 w ，高度为 h

习题

5.1 从你认识的某个人的一张照片开始，对它做一些特定的颜色调整：

- 将牙齿变成紫色。
- 将眼睛变成红色。
- 将头发变成橙色。

当然，如果你朋友的牙齿已经是紫色的，眼睛已经是红色的，或者头发已经是橙色的，

那就换一种目标颜色。

- 5.2 编写一个程序checkLuminance，输入红、绿、蓝三个值，使用加权平均法计算亮度。然后基于计算的亮度结果向用户打印一条警告信息：
 - 如果亮度小于10，那么输出：“太暗了。”
 - 如果亮度在50~200之间，那么输出：“看上去处在让人舒服的范围。”
 - 如果亮度超过250，那么输出：“接近白色了！”
- 5.3 尝试某个范围内的色键处理——从背景中抓出一样东西，而这样东西只位于图片的某一部分。比如，给某人的脑袋加一圈光环，但不要与身体的其他部分掺杂起来。
- 5.4 编写一个函数融合两幅图片，先从第一幅图片顶部的1/3开始，然后把两幅图片中间的1/3融合起来放在中间，最后显示第二幅图片底下的1/3。两幅图片大小相同时效果最好。
- 5.5 如果图片上的某个像素与其右侧像素的色差超过某个输入值，那么就在图片上画一条黑线。
- 5.6 编写一个函数把两幅图片交织起来。首先从第一幅图片中取20个像素，然后从第二幅图片中取20个像素，然后再从第一幅图片中取后面的20个像素，然后再从第二幅图片中取后面的20个像素，依此类推，直到用完所有的像素。
- 5.7 编写一个函数，取一幅图片的25%与另一幅图片的75%融合。
- 5.8 只用一重循环重写编码函数。
- 5.9 通过在图片上绘制文本来制作一张电影海报。
- 5.10 将三、四幅图片横向拼在一起，再添加一些文本，制作一张连环漫画。
- 5.11 使用绘图函数画一只牛眼。
- 5.12 使用本章介绍的绘图工具画一栋房子——只要简单的玩具房子就可以，有一房门，两个窗户，有墙和屋顶。
- 5.13 在一张图片上画横线和竖线，线条之间相隔10个像素。
- 5.14 在一张图片上画左上角到右下角的对角线。
- 5.15 在一张图片上画右上角到左下角的对角线。
- 5.16 什么是基于向量的图像？它与位图图像有何不同？什么情况更适合使用基于向量的图像？
- 5.17 在沙滩上之前我们放置神秘盒子的地方画一栋房子。
- 5.18 写一个函数画一张简单的脸，要有眼睛和嘴巴。
- 5.19 为什么电影制作人使用绿色或蓝色幕作特效而不使用红色幕？
- 5.20 找一块绿色的广告纸板，让一个朋友站在它前面拍张照片。现在，通过色键技术把他放到丛林中，或放到巴黎去。
- 5.21 用画房子函数画一座小城镇，镇上有十几栋大小不一的房子。你可以考虑修改一下画房子的函数，把房子画在一个输入坐标指定的位置，然后画每一栋房子时分别修改一下坐标。
- 5.22 画一道彩虹——使用你所知道的有关颜色、像素和绘图操作的知识来画一道彩虹。使用绘图函数更方便还是操作单个像素更方便呢？为什么？

深入学习

John Maeda现在在MIT媒体实验室的美学与计算小组（Aesthetics and Computation Group）工作。你可以看看他的媒体处理环境（<http://www.processing.org>），这是一套面向交互式艺术、活动视频处理和数据可视化开发的环境。它同样能完成本章实现的一些效果。

声 音

第6章 使用循环修改声音

第7章 修改一段样本区域

第8章 通过合并片段制作声音

第9章 构建更大的程序

使用循环修改声音

本章学习目标

本章媒体学习目标：

- 理解声音是如何数字化的，理解使我们得以将声音数字化的人类听觉局限。
- 使用奈奎斯特定理（Nyquist theorem）确定声音数字化所需的采样率。
- 处理音量。
- 创建（和避免）削波。

本章计算机科学学习目标：

- 理解并使用数组这一数据结构。
- 基于 n 位共有 2^n 种可能的模式，确定保存数值所需的位。
- 使用声音对象。
- 调试声音程序。
- 使用迭代（for循环）处理声音。
- 基于作用域理解变量何时可用。

6.1 声音是如何编码的

要理解声音编码和处理的方式，需要理解两部分内容：

- 首先，声音的物理机制是什么？我们是如何听到各种不同声音的？
- 其次，我们是如何将声音映射为计算机中的数字的？

6.1.1 声音的物理学

从物理上讲，声音是气压形成的波。一件东西发出声音时，它实际是在大气中引起了涟漪，就像石头或雨滴落入池塘时在水面引起涟漪一样（如图6.1所示）。每一滴水都会引起水压的波沿水面四散开来，从而在水中引起可见的上涨，以及不那么容易看见，但规模相同的下落。上涨是水压的升高，下落是水压的降低。我们看到的涟漪有些实际来自于涟漪的组合——有些波是其他波的累加和交互。

在空气中，我们将这种压力的升高称为密部（compression），压力的降低称为疏部（rarefaction）。正是这些密部和疏部使我们能听到声音。波的形状、频率（frequency）和振幅（amplitude）都会影响我们对声音的感知。

世界上最简单的声音是正弦波（sine wave，见图6.2）。在正弦波中，密部和疏部按相同的幅度和频度到来。正弦波中一个密部加一个疏部称为一个周期（cycle）。在周期内的某个时刻，必然有一个位于密部和疏部之间的压力为零的点。零点到最大压力点的距离称为振幅。

一般来讲，振幅是影响我们感知音量（volume）最重要的因素：振幅升高，通常我们会感觉到声音变大。当我们感知到音量增加时，我们说感觉到了声间强度（intensity）在增加。

人类对声音的感知并不是物理现实的直接映射。对人类感知声音的研究称为心理声学 (psychoacoustics)。关于心理声学，一个奇特的事实是：我们对声音的感知与真实现象之间多是对数关系。

我们用分贝 (decibel, dB) 衡量强度的变化。这大概是最能让你跟音量联系起来的单位。分贝是一种对数度量，因此与我们感知音量的方式相符。它永远是个比率，是两个值的比较。 $10 \cdot \log_{10}(I_1/I_2)$ 表示的就是两个声强 I_1 和 I_2 之间以分贝表示的强度变化。如果基于相同的条件来衡量两个振幅，那么我们可以表达同样的定义： $20 \cdot \log_{10}(A_1/A_2)$ 。如果 $A_2 = 2 \cdot A_1$ (即振幅加倍)，那么其差约为 6 dB。



图6.1 雨滴在水面引起的涟漪，正如声音在空气中引起的涟漪

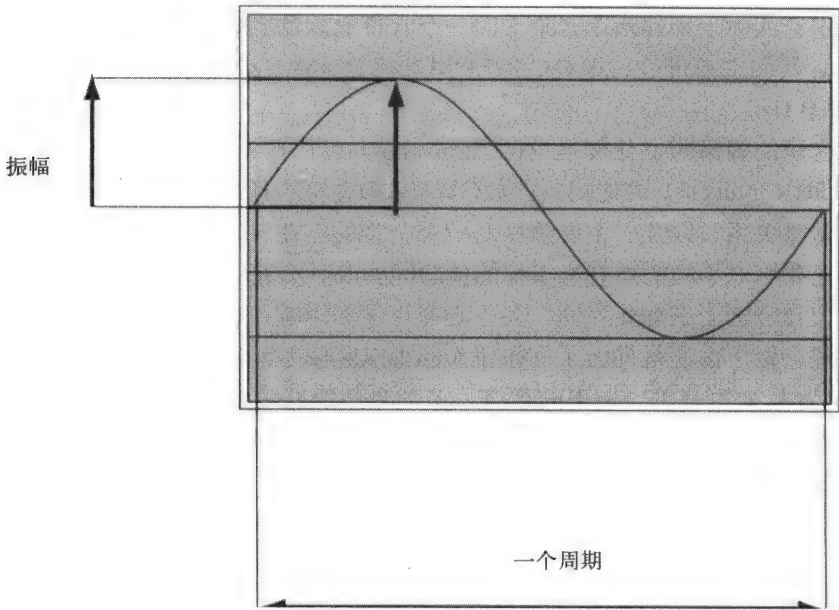


图6.2 最简单的声音：正弦波的一个周期

分贝用作绝对度量时，它相对的是声压级 (Sound Pressure Level, SPL) 的可听度阈值：0 dB SPL。日常交谈的声强大约在 60 dB SPL。大声交谈大约在 80 dB SPL。

反映周期发生频度的度量值称为频率 (frequency)。如果周期较短，那么每秒钟就会有 很多周期。如果周期较长，每秒钟的周期数就会少一些。频率增加时，我们会感知音高 (pitch)

在上升。我们用每秒周期数 (Cycles Per Second, CPS) 度量频率, 也称为赫兹 (Hertz或Hz)。

一切声音都是周期性的——总会有某种疏部和密度的模式构成周期。在正弦波中, 周期的概念很明了。在自然波中, 模式重复的形式没有这么清晰。即使在池塘的涟漪中, 波也不见得像你想象的那么规律。两个波峰之间的时间并不总是一样长——它是变化着的。这意味着一个周期可能包含多个波峰和波谷, 直到模式重现。

人类能听到20 Hz到20 000 Hz (或20 kilohertz, 简写为20 kHz) 之间的声音。这又是一个很大的范围, 与振幅类似。举个例子可以让你对音乐在频谱中的分布有个感性认识: 在传统的平均律 (equal temperament) 定音中, 中央C之上的A音符是440 Hz (如图6.3所示)。

与强度类似, 我们对声音的感知几乎与频率的对数恰好成正比。关于音高, 我们感知到的并不是频率的绝对差而是其比率。如果听到100 Hz的声音, 接着又听到200 Hz的, 那么你感觉到的音高变化与1000~2000 Hz的变动是一样的。100 Hz的差值显然远远小于1000 Hz的变化, 但我们的感觉却是一样的。



图6.3 中央C之上的A音符是440 Hz

在标准定音法中, 相邻两个音阶上同一个音符之间的频率比为2:1。每提高一个音阶, 频率增加一倍。我们之前讲过, 中央C之上的A音符是440 Hz, 根据这一规律, 你就能知道再后面一个A是880 Hz。

我们对音乐的理解与文化标准 (cultural standard) 有关, 但其中还是有一些共通的东西, 包括音程 (pitch interval) 的使用 (比如C音符和D音符的频率比值在每个音阶中都是一样的), 音阶之间的常量关系, 以及一个音阶中4~7个全音的存在 (这里不考虑升半音和降半音)。

是什么让我们对不同的声音有不同的体验呢? 为什么笛子演奏的某个音符跟小号或单簧管演奏的同一个音符听上去如此不同? 关于心理声学和影响声音感知的物理特性, 目前我们还没有完全搞清楚, 但下面还是列出了一些使人们得以区分不同声音 (特别是乐器声音) 的因素:

- 真实声音从来不是单一频率的声波。多数自然声音中包含多种频率, 通常各自的振幅也不一样。这些额外的频率通常称为泛音 (overtone)。比如, 钢琴演奏C音符的时候, 丰富的间质中也包含E和G两种音符的声音, 只不过振幅小一些。不同的乐器在演奏的音符中包含不同的泛音。中心的音调, 也就是我们尝试演奏的那个, 称为基音 (fundamental)。
- 从振幅和频率来看, 乐器声音是不连续的。有些乐器慢慢达到目标频率和振幅 (比如管乐器), 另外一些则很快达到目标频率和振幅, 然后音量逐渐衰减, 频率却保持高度的恒定 (比如钢琴)。
- 并非所有声波都能用正弦波来规则地表示。真实声音中会有些古怪的凹凸和尖利的边沿。我们的耳朵可以听到这些凹凸和边沿, 至少在一开始的几个波中如此。我们可以用正弦波来合成声音, 且效果不错, 而声音合成器有时也采用其他波形来获得不同声音 (如图6.4所示)。

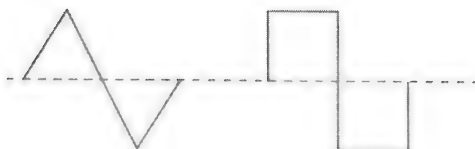


图6.4 有些合成器使用三角波（也叫锯齿波）或方波

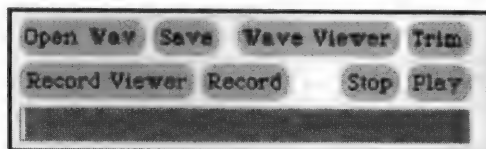


图6.5 声音编辑器工具

6.1.2 探索声音的样子

你可以从<http://www.mediacomputation.org>上面获得MediaTools应用程序，以及它的帮助文档。MediaTools应用包含了用于声音、图形和视频的工具。你可以使用声音工具在声音进入计算机的麦克风时观察它，从而获得一些感性认识：响亮的声音和柔和的声音分别是什么样子的，高亢的声音和低沉的声音分别是什么样子的。

你会看到JES中也有一个MediaTools菜单。JES的MediaTools同样支持声音和图片的查看。但你无法在声音接触计算机的麦克风时实时观察它。而在MediaTools应用程序中，你可以实时地看到声音。

图6.5是MediaTools应用程序的声音编辑器。你可以用它录音，打开磁盘上的WAV文件，或者以不同的方式查看声音。（当然，你的计算机上要有麦克风！）

要查看声音，先点击Record Viewer按钮，然后点击Record按钮。（点击Stop停止录音。）在这个编辑器中，声音可以显示成三种视图。

第一种是信号视图（signal view，见图6.6）。在这一视图中，你看到的是未经处理的声音——每次气压增大会引起图中曲线的上升，每次声压减小则会引起图中曲线的下降。注意波形变化得很快。你可以用一些更柔和或更响亮的声音尝试一下，看看波形的样子是如何变化的。你可以随时点击Signal按钮从其他视图切换回信号视图。

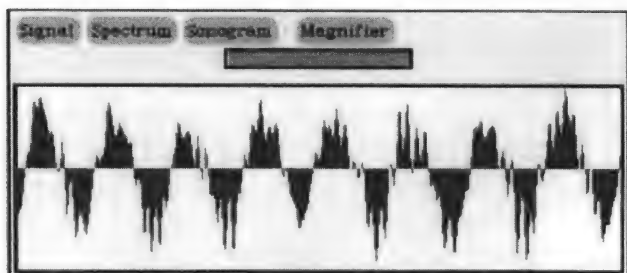


图6.6 随着信号的进入查看声音

第二种视图是频谱视图 (spectrum view, 见图6.7)。频谱视图是一种完全不同的声音视角。上一节我们已经了解到自然声音实际常常由多种不同的频率同时组合而成。频谱视图分别显示了这些频率。这种视图也称为频域 (frequency domain)。

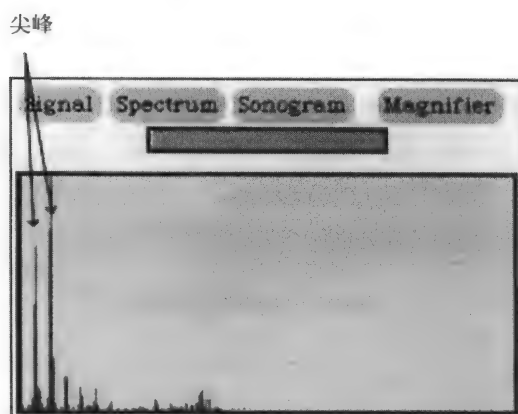


图6.7 在频谱视图中查看有多个尖峰的声音

在频谱视图中, 频率从左向右依次递增, 纵向的高度显示了声音中这种频率的能量大小 (差不多可以理解成音量大小)。自然声音看起来就像图6.7那样具有多个尖峰 (图形中曲线的上升)。(尖峰附近那些更小的上升常被视为噪声。)

产生频谱视图的方法在技术上称为傅立叶变换 (Fourier transform)。傅立叶变换将声音从时域 (声间随时间的升降) 转换到频域 (确定这段时间的声音里含有哪些频率, 以及这些频率对应的能量)。在这一视图中, 频率自左向右依次递增 (左边是低频, 右边是高频), 频率的能量越大对应的尖峰也越高。

第三种视图是声谱视图 (sonogram view, 见图6.8)。声谱视图与频谱视图有一点相像: 它们都描述频域, 但声谱视图呈现的是频率随时间的变化。声谱视图中的每一列, 有时也称为

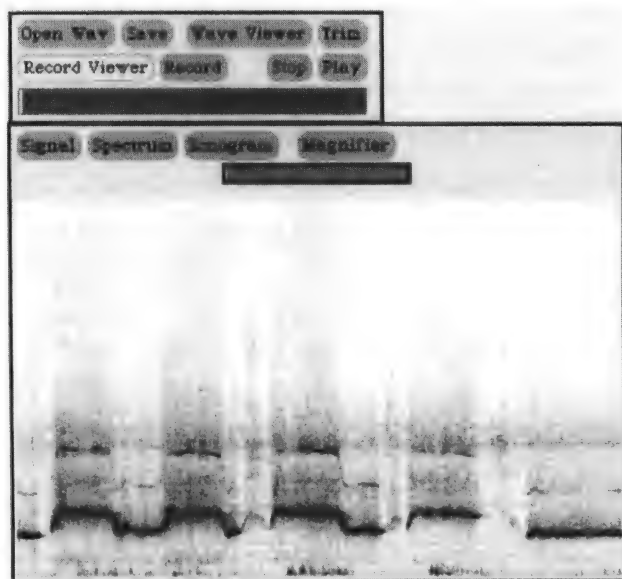
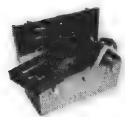


图6.8 在声谱视图中查看声音信号

一个片段或（时间的）窗口，呈现了给定时刻的所有频率。片段中的频率从低（底部）向高（顶部）增大。因此，图6.8表示了一段声音从低走高又走低的过程。同一列中各位置的黑暗程度显示了输入声音中该频率在给定时刻的能量大小。声谱视图最适于研究声音随时间的变化（比如，弹某个琴键，它的声音是如何随着音符的衰减而变化的，不同乐器的声音有怎样的不同，不同的说话声又是如何不同的）。



实践技巧：查看声音

你绝对应该基于真正的声音看看这些不同的视图，这样，你会对声音本身，以及本章的操作对声音做了什么有更好的理解。

6.1.3 声音编码

你刚刚读完了声音的物理原理以及我们对声音的感知方式。为了在计算机上处理声音并回放它们，我们必须将声音数字化。声音数字化意味着接受这种声波流并将它转化为数字。我们想要的是：抓取一段声音，可能对它做些操作，然后回放它并尽可能精确地听到我们抓取的声音。

声音数字化过程的第一步由计算机硬件，即计算机上的物理设备完成。只要计算机上有麦克风和相应的声音设备（比如Windows计算机上的SoundBlaster声卡），就可以随时将麦克风受到的气压度量为一个数字。正数对应气压的上升，负数对应气压的疏部。我们把这一过程称为模数转换（Analog-to-Digital Conversion, ADC）——我们把一个模拟信号（持续的声波变化）转成了一个数字值。这意味着我们可以获得声压的瞬时度量值，但这只是第一步。声音是持续变化的声波。我们如何在计算机中保存它呢？

顺便提一下，计算机上声音回放系统的工作原理基本上就是把前面的过程反过来。声音硬件进行数模转换（Digital-to-Analog Conversion, DAC），然后把模拟信号发送到扬声器。DAC过程也需要代表压力的数字。

如果你懂微积分，就能大致想到我们的做法。你知道，我们可以用很多高度与曲线一致的矩形来近似地度量曲线下的面积（如图6.9所示），矩形越多结果就越接近。基于这一思想，很显然如果我们能捕捉足够多的麦克风压力读数，就等于捕捉了声波。我们把每个压力读数称为一个样本（sample）——就是在“采集”那一时刻的声音样本。可我们需要多少样本呢？在积分学中，我们通过（概念上的）无限多个矩形来计算曲线下的面积。可是，虽然计算机的内存一直在变大，但我们还是无法每秒钟捕捉无限次样本。

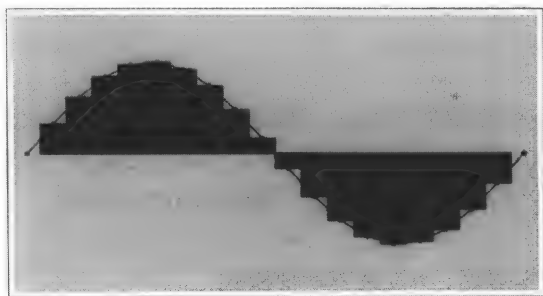


图6.9 用矩形估算的曲线下面积

数学家和物理学家早在计算机出现之前就开始考虑这些问题了，至于我们需要多少个样本，

答案也在很早之前就计算好了。答案取决于你想捕捉的最大频率。假定你不关心8000Hz以上的声音，奈奎斯特定理（Nyquist theorem）指出：我们需要每秒捕捉16 000次样本才能完全捕捉并定义一段频率低于每秒8000周期的声波。

计算机科学思想：奈奎斯特定理

要捕捉每秒钟最多 n 个周期的声音，你需要每秒钟捕捉 $2n$ 次样本。

这不仅是理论上的结果，奈奎斯特定理已经用在了我们的日常生活中。事实证明，人类的语音通常不会超过4000 Hz。这就是为什么我们的电话系统设计成了每秒钟大约8000次采样。这也是为什么通过电话播放音乐效果并不好。（大多数）人的听力极限在22 000 Hz，如果每秒钟捕捉44 000次样本，我们便能捕捉到任何一种可以实际听到的声音。制作CD时每秒钟的采样数是44 100次——比44 kHz多一点，因为一些技术原因和一个附加因子（fudge factor）。

我们把采集样本的速度称为采样率。日常生活中听到的大多数声音的频率都远低于我们的听力极限。对于这类声音，你可以用22 kHz（每秒钟22 000个样本）的采样率来捕捉并操控它，效果已经相当不错。如果使用太低的采样率来捕捉音调很高的声音，回放时你还是可以听到声音，但音高听起来会很怪异。

通常，每次采样都用2字节，或16位来编码。虽然更大的样本容量（sample size）也存在，但16位对大多数应用来说已经非常好了。CD质量的声音采用的就是16位样本。

6.1.4 二进制数和二进制补码

16位可以编码的数字范围是-32 768~32 767。这些数字不是凭空而来的，如果你理解这种编码，就会明白它们有确切的来历。这些数字使用一种称为二进制补码表示法（two's complement notation）的技术在16位中进行编码，但我们不需要了解这种技术的细节也同樣能理解这些数字的来历。一共有16位表示负数和正数，让我们留出一位（记住，它只能是0或1）来表示正（0）和负（1），并把这一位称为符号位。这样就只剩下15个位来表示实际的值。15位共有多少种不同模式呢？我们可以数一数：

```
0000000000000000
0000000000000001
0000000000000010
0000000000000011
...
1111111111111110
1111111111111111
```

这看起来真不幸。我们看看能否找出一种模式。2位有4种模式：00、01、10、11。3位有8种模式：000、001、010、011、100、101、110、111。 2^2 是4， 2^3 是8。试一下4位，有多少种模式呢？ $2^4 = 16$ 。事实证明，我们可以把这一规律作为普遍原理。

计算机科学思想： n 位共有 2^n 种模式

如果你有 n 位，那它们可以组成 2^n 种不同模式。

$2^{15} = 32\,768$ 。为什么负数比正数多一个呢？原因在于0是非正非负的，要把它表示成位，

我们就需要把某种模式定义为0。我们用一个正数范围（符号位为0）的值来表示0，结果它把32 768种模式占掉了一个。

计算机常用的表示正整数和负整数的方法称为二进制补码。在二进制补码中，正数与常规的二进制表示一样。数字9的二进制表示为00001001。负数的补码则可以这样计算：首先从相应正数的二进制表示开始，然后把它按位取反，让1变成0，0变成1，最后把结果加1。因此，要计算-9的补码，我们从9的补码00001001开始，把它按位取反后变为11110110，然后加1得结果11110111。使用二进制补码表示数字的优势之一在于：如果把一个负数（比如-9）跟绝对值相同的正数（9）相加，结果会是0，因为1加1得0进1（如图6.10所示）。

$$\begin{array}{r}
 1111111 \\
 \hline
 00001001 \\
 + 11110111 \\
 \hline
 00000000
 \end{array}$$

图6.10 9和-9的补码相加

6.1.5 存储数字化的声音

样本容量限制了可以捕捉的声音振幅。如果一种声音能产生大于32 767的压力（或低于-32 768的疏部），那么我们也只能捕捉到16位的极限。这种情况下，如果从信号视图中查看波形，看上去就像有人拿剪刀剪掉了波峰部分。因为这个原因，我们把这种效应称为削波（clipping）。播放（或产生）削波的声音效果会很差——就像扬声器坏了一样。

声音数字化还有其它方法，但这一种是迄今为止最常见的了。这种编码声音的方法在技术上称为脉冲编码调制（Pulse Coded Modulation, PCM）。或许你以前见过这个术语，如果你读过更多关于音频的资料或摆弄过音频软件的话。

这意味着声音在计算机中是一长串的数字，每个数字都是某时刻的样本。这些样本是有序的：如果不按正确的顺序播放它们，那么就无法得出与原来一样的声音。在计算机上保存数据项目的有序列表，最高效的方法就是使用数组（array）。数组在内存中是一串连续的字节。我们把数组中的每个值称为一个元素（element）。

将组成声音的样本保存在数组中非常容易。想象一下，每两个字节存储一个样本。数组将会很大——对于CD质量的声音，每秒钟的录音会有44 100个元素。这样，一分钟的录音会产生2 646 000个元素的数组。

每个数组元素都有一个称为下标的数字与之关联。下标数字的顺序依次递增。数组中第一个元素下标为0，第二个元素下标为1，依此类推。最后一个元素的下标等于数组中的元素个数减1。你可以把数组想象成长长的一列盒子，每个盒子都持有一个值且具有一个下标数字（如图6.11所示）。

你可以用媒体工具查看一段声音（如图6.12所示），看一看哪里是寂静（小振幅）的，哪里是响亮（大振幅）的。这对于声音的处理很重要。例如，语音录音中两个单词之间的间隙常常是寂静的——至少比单词本身寂静。通过寻找间隙，你可以辨认出每个单词的结尾，如图6.12所示。

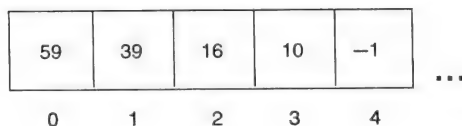


图6.11 一段真实声音中的前5个元素

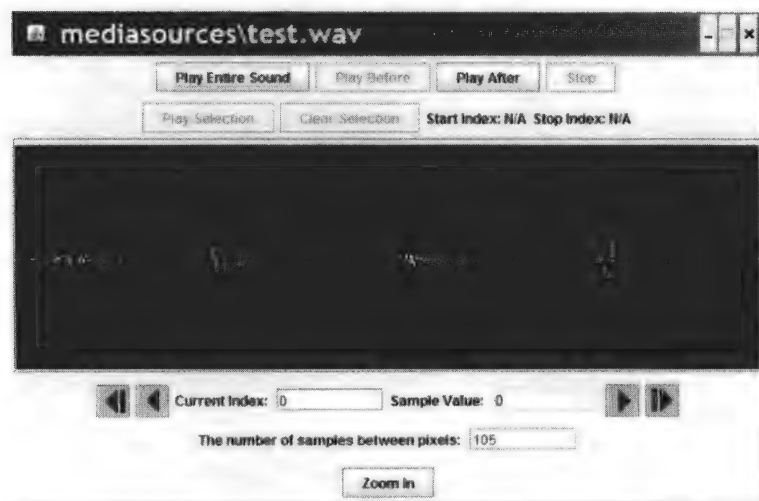


图6.12 媒体工具中图示的录音

本章后面很快会涉及以下内容：如何读取一个包含录音的文件并构造一个声音对象。如何查看声音中的样本并修改声音数组元素的值。改变了数组中的值，也就改变了声音。对声音的处理无非就是对声音数组元素的处理。

6.2 处理声音

了解了声音的编码方式，我们就可以用Python程序来处理声音了。以下是我们需要做的事情：

1. 我们需要得到一个WAV文件的名称并基于它构造一个声音对象。
2. 你会经常获取声音样本。样本对象很容易处理，而且它们能“知道”你做的改动，从而改变它们也就自动改变了原来的声音。首先是如何处理样本，这是起点，然后会看到如何在声音中处理样本。
3. 不论是从声音中取出样本对象，还是直接处理声音对象中的样本，接下来都要对样本做一些操作。
4. 查看原始声音和修改过的声音，从而检查结果是否与你期望的相符。
5. 把声音回写到新的文件中，以便在其他地方使用。

6.2.1 打开声音并处理样本数据

使用pickAFile，你可以选取一个文件并获得它的完整路径名，然后使用makeSound构造一个声音对象。以下的例子是JES中的做法。

```
>>> filename=pickAFile()
```



```
>>> print filename
C:/ip-book/mediasources/preamble.wav
>>> aSound=makeSound(filename)
>>> print aSound
Sound file: C:\ip-book\mediasources\preamble.wav
number of samples: 421110
```

makeSound所做的是基于输入的文件名从文件中捞起所有字节，把它们放进内存中，并打上一个个大大的标记，宣称“这是一段声音！”执行aSound = makeSound(filename)就等于说：“把那个声音对象命名为aSound！”把声音作为输入传给其他函数就等于说：“把那个声音对象（是的，就是名叫aSound的那个）用作这个函数的输入。”

你可以用getSamples获得声音的样本。getSamples函数接受一个声音对象作为输入并返回样本对象的数组。这个函数执行时，可能需要很长的时间才能完成——声音越长时间越长，声音越短时间越短。

getSamples函数基于基础样本数组构造样本对象的数组。对象不仅仅是你前面了解到的样本值——区别之一是，样本对象还知道它来自于哪个声音对象以及它的下标。后面你会读到更多关于对象的内容，目前只按表面意思理解即可：getSamples为你提供了一串可供处理的样本对象——实际上，处理它们非常容易。你可以用getSampleValue（使用一个样本对象作为输入）来获得样本对象的值，使用setSampleValue（使用一个样本对象和一个新值作为输入）来设置样本值。

但在开始处理样本之前，我们先看看其他一些获得或设置样本值的方法。我们可以用getSampleValueAt函数让声音对象给出特定下标处的特定样本值。getSampleValueAt函数的输入值是一个声音对象和一个下标数字。

```
>>> print getSampleValueAt(sound,0)
36
>>> print getSampleValueAt(sound,1)
29
```

0到声音样本长度减1之间的任何整数都是合法的下标值（0.1289则不是一个好下标）。我们可以用getLength()来获得样本长度。注意，如果我们试图取得超出数组结尾的样本，会得到如下的错误信息：

```
>>> print getLength(sound)
421110
>>> print getSampleValueAt(sound,421110)
You are trying to access the sample at index: 421110,
but the last valid index is at 421109
The error was:
Inappropriate argument value (of correct type).
An error occurred attempting to pass an argument
to a function.
```



调试技巧：获得与错误有关的信息

如果程序出现错误而你想得到更多信息，那么可以点击JES Edit菜单中的Options项，然后选择Expert模式而不是Normal模式（如图6.13所示）。Expert模式有时能提供更多细节——可能比你想知道的还多，而且可能多次提供同一项内容，但它有时大有裨益。

类似地，我们可以用`setSampleValueAt`来修改样本值。它也接受一个声音对象、一个下标，但还接受该下标处的一个新样本值。我们可以用`getSampleValueAt`再检查一遍。

```
>>> print getSampleValueAt(sound,0)
36
>>> setSampleValueAt(sound,0,12)
>>> print getSampleValueAt(sound,0)
12
```

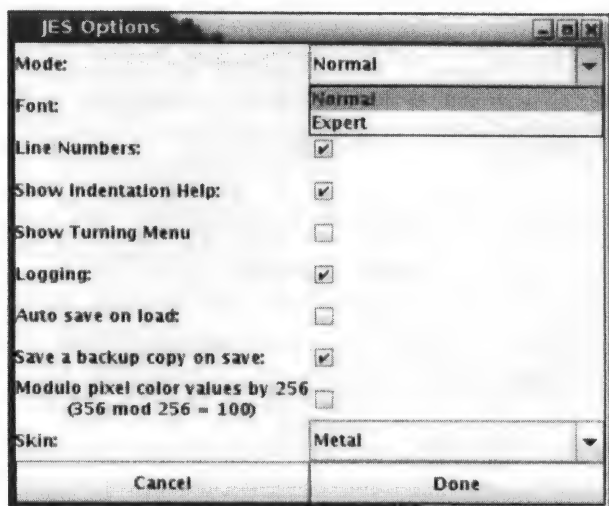


图6.13 打开Expert错误模式



常见bug：名字拼写错误

刚才你见到了许多函数名字，有一些还很长。如果拼错了某个名字会怎样呢？

JES会抱怨它不明白你的意思，就像这个：

```
>>> writeSndTo(sound,"mysound.wav")
Name not found globally.
A local or global name could not be found. You need to
define the function or variable before you try to use
it in any way.
```

这不是大问题，你可以用键盘上的上箭头键调出最后输入的内容，然后通过左箭头键移出错误位置并改正错误。要确保光标回到行末再敲回车，可以用右箭头键移过去。

如果播放这段声音，你觉得会是什么效果？我们已经把第一个样本的值从36改成了12，听起来真的会跟之前不一样吗？实际上不会。要解释为什么，让我们先用`getSamplingRate`函数找出这段声音的采样率，此函数接受一个声音对象作为输入。

```
>>> print getSamplingRate(sound)
22050.0
```

在这个例子中，我们处理的声音（Mark的录音，朗读美国宪法前言中的一部分）采样率为每秒22 050次。改变一个样本只改变了第一秒声音的1/22 050。如果你能听出这种区别，那你的听力真是好得令人吃惊——而我们会怀疑你说的是否真实。

显然，要对声音做显著的处理，即使不处理上千个样本，也要处理几百个样本。当然，我

们不会像下面这样键入上千行命令来实现这一目标。

```
setSampleValueAt(sound,0,12)
setSampleValueAt(sound,1,24)
setSampleValueAt(sound,2,100)
setSampleValueAt(sound,3,99)
setSampleValueAt(sound,4,-1)
```

我们需要充分利用执行菜谱的计算机，告诉它把某事做几百遍甚至上千遍。这是下一节的主题。

在这一节结束之前，我们将讨论一下如何把结果回写到文件中。一旦处理了声音并打算保存结果以备别处使用时，你就需要使用writeSoundTo，它接受一个声音对象和一个新的文件名作为输入。既然保存的是声音，确保文件名以“.wav”扩展名结尾，这样操作系统才会知道如何正确对待它。

```
>>> print filename
C:/ip-book/mediasources/preamble.wav
>>> writeSoundTo(sound,"C:/ip-book/mediasources/new-preamble.wav")
```

多次播放声音时你会发现：如果连续快速地多次使用play，声音会混合起来。第二次play在第一次结束之前开始播放。如何保证计算机只播放一段声音，然后一直等到它结束呢？你可以使用blockingPlay，它的功能与play一样，但它会等待声音播放结束，因此在播放时不会有其他声音干扰。

6.2.2 使用JES媒体工具

JES媒体工具(MediaTools)可以从JES的MediaTools菜单中使用。当你选择图片或声音工具时，会显示与这种工具对应的弹出式菜单(如图6.14所示)，其中包含图片变量或声音变量。点击OK就会进入JES声音工具。也可以通过查看声音来调出声音工具，就像查看图片时那样。

```
>>> explore(sound)
```

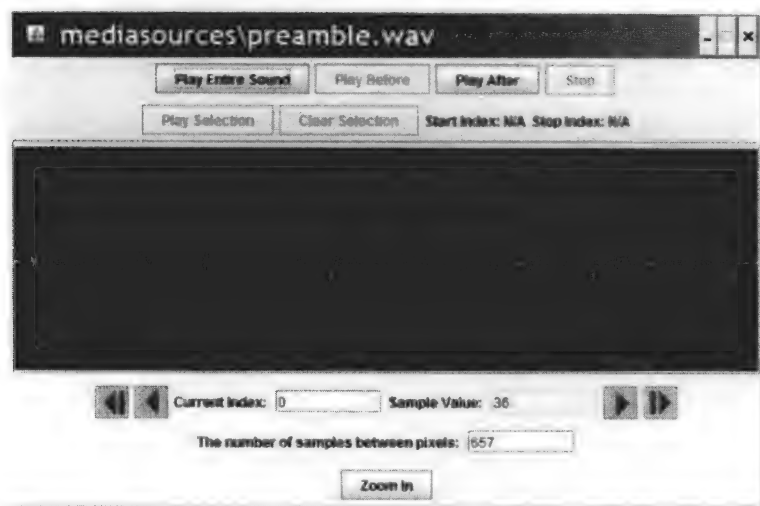


图6.14 在JES中查看声音

声音工具使你能查看一段声音。

- 你可以播放声音，点击其中的任何地方来标记一个位置，然后播放标记之前或之后的部分。
- 你可以（通过点击和拖拽）选择一块区域，然后只播放这一区域（如图6.15所示）。
- 设置标记时，工具中会显示样本下标以及该点的样本值。
- 你还可以放大显示，从而看清每个声音值（如图6.16所示）；你必须拖动窗口的滚动条才能看到所有的值。

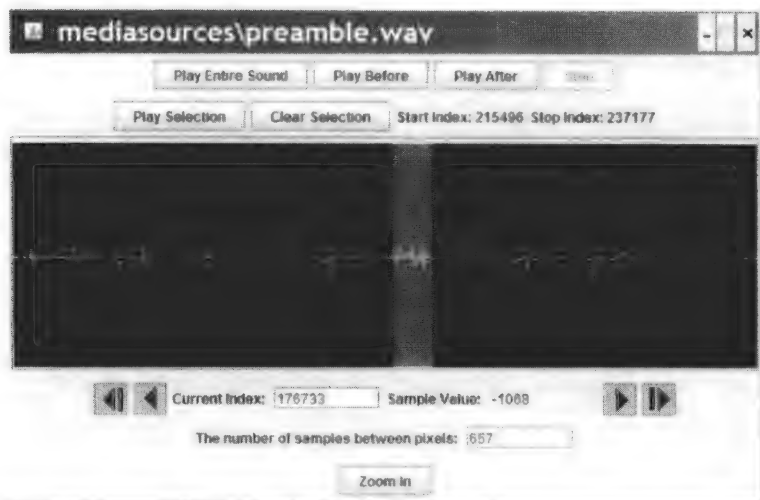


图6.15 在JES中查看声音



图6.16 放大显示，看清每一个声音值

常见bug：Windows和WAV文件



WAV文件的世界并不如你想象地那般兼容与畅通。使用其他应用程序（比如Windows录音机）创建的WAV文件可能无法在JES中播放，而JES的WAV文件也不一定能在所有其他应用程序（比如WinAmp 2）中播放。Apple QuickTime Player Pro (<http://www.apple.com/quicktime>) 的一个强项便是：它能读入任何WAV文件并输出一个几乎能被其他任何应用程序读取的新文件。

6.2.3 循环

我们面对的是计算机科学中的一个普遍问题：如何让计算机反反复复地做一件事？我们需要让计算机循环或迭代（iterate）。Python有专门用于循环（或迭代）的命令。大多数情况下，我们用for命令。for循环针对（你提供的）一个序列执行（你指定的）命令，每次命令执行时，（你命名的）一个变量会持有序列中不同元素的值。

6.3 改变音量

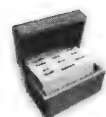
先前我们讲过，声音的振幅是音量的主要因素。这意味着增加振幅就可以提高音量，减少振幅就可以降低音量。

有一点不要混淆——改变振幅并不会使计算机自动旋转扬声器的音量旋钮。如果你调低了扬声器的音量（或计算机的音量），那么声音无论怎么调也不会特别响亮。这一节的目的是让声音本身变得更响。你有没有经历过这种情况：在电视上看一部影片，你并没有改变电视机的音量，声音却突然变得很低，低到你几乎听不见？（我想起了电影《The Godfather》中Marlon Brando说话时的声音。）或者，你有没有注意到商业广告总是比常规电视节目的声音更响？这就是我们这一节要做的事情。我们通过调整振幅，可以让声音或如咆哮，或似低语。

6.3.1 增大音量

让我们把声音中每个样本的值增加一倍，以此来提高音量。我们可以用getSamples来得到声音中的样本序列（或数组），用for循环遍历序列中的所有样本。针对每个样本，我们取出它的当前值，然后把它设为当前值的两倍。

下面就是将输入声音的振幅加倍的函数。



程序54：通过振幅加倍来提高输入声音的音量

```
def increaseVolume(sound):
    for sample in getSamples(sound):
        value = getSampleValue(sample)
        setSampleValue(sample, value * 2)
```

把上面的代码输入到JES程序区。点击Load Program让Python处理这个函数，从而我们可以使用increaseVolume这个名字。试着效法下面的例子来深入了解它的全部原理。

为使用这一菜谱，你必须首先创建一段声音，然后把声音作为输入传给increaseVolume。在下面的例子中，我们获得文件名的方法是把变量f显式设为一个文件名字符串，而不是使用pickAFile。别忘了你不太可能原封不动地输入这段代码就让它正常运行，你的文件名可能和我的不一样。

```
>>> f="C:/ip-book/mediasources/test.wav"
>>> s=makeSound(f)
>>> explore(s)
>>> increaseVolume(s)
>>> explore(s)
```

我们创建了声音对象并命名为s。然后，我们查看声音对象，这一命令会基于声音对象的副本调出JES声音工具。接下来，我们计算increaseVolume(s)，名为s的声音在那个函数内部

又被命名为sound。这是非常重要的一点。两个名字引用的是同一段声音！increaseVolume中所做的改变当然会改变同一段声音。你可以认为两个名字互为别名：它们引用了同样的东西。

还有一点，这里只是顺带提一下，后面会很重要：increaseVolume函数结束后，名字sound将不再有值。它只存在于函数执行的过程中。我们说：它只存在于函数increaseVolume的作用域中。变量的作用域就是程序知道它的那个范围。命令区定义的变量具有命令区作用域，只有命令区知道它们。函数中定义的变量具有函数作用域，只有函数内部知道它们。

现在，我们可以播放文件，听听是不是更响了，然后把它写到一个新文件中。

```
>>> play(s)
>>> writeSoundTo(s, "c:/ip-book/mediasources/test-louder.wav")
```



常见bug：让声音尽量短

更长的声音会占用更多内存，处理起来也会更慢。

6.3.2 真的行吗

现在，它是真的更响了，还是仅仅貌似更响呢？我们可以用多种方法来检验一下。肯定可以不断把声音变得更响，方法就是对声音多执行几次increaseVolume——最终，你完全可以确信声音是真的变响了。但也存在一些方法可以检验更微妙的效应。

如果用JES媒体工具中的声音工具来比较两段声音的图形，就会发现，经过使用函数增加之后，声音图形确实具有更大的振幅。图6.17中可以看出来。

可能你还是不敢确定第二幅图中看到的是更大的波形。可以用JES声音工具来检查单个样本值。在一个声音工具窗口的波形中点击一个具有非0值的位置，然后在另一个声音工具窗口中输入它的下标并按回车。比较一下样本值。你看，图6.17中下标为18 375的值原来是-1 290，增加音量之后它变成了-2 580，可见，函数确实把样本值加倍了。

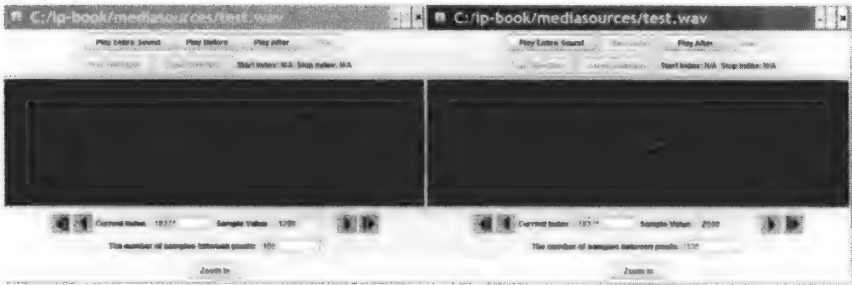


图6.17 比较原始声音（左）和更响亮版本（右）的图形

最后，肯定可以在JES中手工检查。如果按照这个例子做过^①，那么变量s现在就是更响亮的声音。f应该还是原来声音的文件名。现在基于f构造一个新的声音对象，它就是原先的声音——在下面的例子中被命名为sOriginal (sound original)。你可以随便检查任何一个样本值。响亮声音的样本值是原来声音的两倍，这一点肯定成立。

① 什么？你没有？你应该做一遍！只有亲自尝试过，它才更有意义。

```

>>> print s
Sound file: C:/ip-book/mediasources/test.wav
number of samples: 67585
>>> print f
C:/ip-book/mediasources/test.wav
>>> sOriginal=makeSound(f)
>>> print getSampleValueAt(s,0)
0
>>> print getSampleValueAt(sOriginal,0)
0
>>> print getSampleValueAt(s,18375)
-2580
>>> print getSampleValueAt(sOriginal,18375)
-1290
>>> print getSampleValueAt(s,1000)
4
>>> print getSampleValueAt(sOriginal,1000)
2

```

可以看出，负值变得更负了。这便是“增加振幅”的含义。波的振幅同时存在于两个方向。我们必须在正维度和负维度上都让波变得更大。

把刚刚读到的内容做一遍，这很重要：要怀疑自己的程序。它真的完成了你想要的动作吗？检查的方法是测试。这是本节要讲的。刚才，你已经看到好几种测试方法：

- 检查结果的一部分（使用JES声音工具）。
- 另外编写代码检查原程序的结果。

程序原理

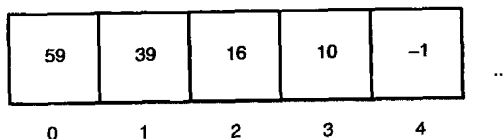
让我们慢慢把代码过一遍，同时考虑它的工作原理。

```

def increaseVolume(sound):
    for sample in getSamples(sound):
        value = getSampleValue(sample)
        setSampleValue(sample,value * 2)

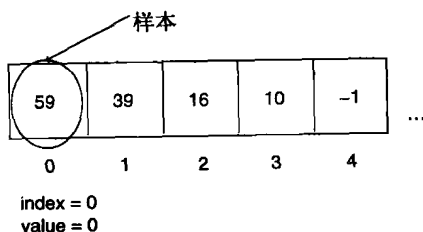
```

还记得描绘声音样本数组的那幅图片吗？下面的图显示了声音对象的前面几个值，这个对象是基于gettysburg.wav文件构造出来的。

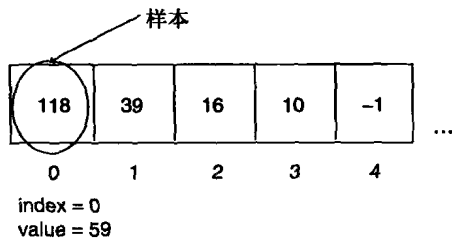


`getSamples(sound)`返回的就是这样的结果：元素编了号的样本值数组。`for`循环允许我们一次一个样本遍历整个数组。名字`sample`将依次赋予各个样本。

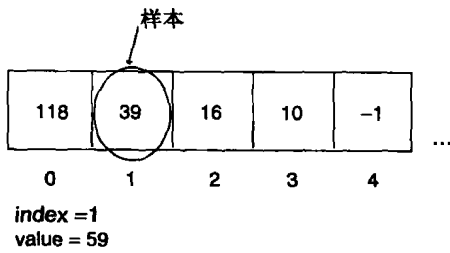
`for`循环开始时，`sample`将成为第一个样本的名字。



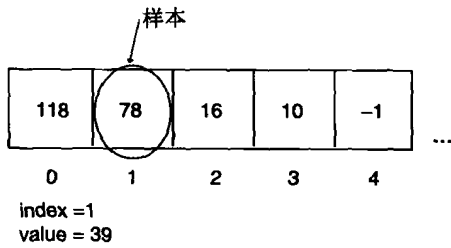
执行到`value=getSampleValue(sample)`时, 变量`value`接受了数值59。然后, 通过`setSampleValue(sample, value * 2)`, 名字`sample`引用的样本变成了原来的两倍。



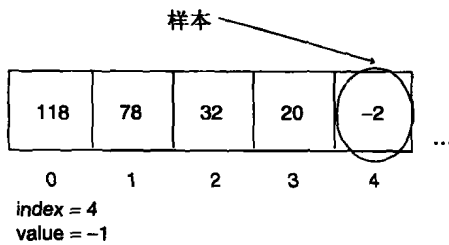
到此循环体的第一次执行结束了。然后Python再次启动循环并移动`sample`, 使它指向数组中的下一个元素。



再一次, 样本的值赋给了`value`, 然后样本变成了原来的两倍。



以下是循环体迭代5次之后样本的样子。



`for`循环继续遍历所有样本——成千上万的样本! 谢天谢地, 执行菜谱的是计算机!

程序中发生的事情也可以这么理解: 从改变声音样本的意义上, `for`循环实际上没做什么。完成工作的是循环体。`for`循环告诉计算机做什么。它是个管理器。计算机实际完成的动作大体是这样的:

```
sample = sample #0
value = sample的值: 59
```



```

样本值变成118
sample = sample #1
value = 39
样本值变成78
sample = sample #2
...
sample = sample #4
value = -1
样本值变成-2
...

```

for循环只说了句“针对数组中的每个元素做所有这些动作”。循环体中包含的才是被执行的Python命令。

上面你刚刚读到的部分称为程序的**跟踪** (trace)。我们慢慢考察了程序执行的每一步，并画图描述程序中的数据。我们使用了数字、箭头、公式，甚至直白的语言来解释程序中发生的动作。这是编程中最重要的技能，也是程序调试 (debugging) 的一部分。你的程序不会一直正常工作。绝对、肯定、毫无疑问地——你会编写功能不符合预期的代码。但计算机还是会完成某些动作。如何确知它做了什么呢？调试！最强大的调试方法就是跟踪程序。

6.3.3 减小音量

减少音量与前面描述的过程相反。



程序55：通过振幅减半来降低输入声音的音量

```

def decreaseVolume(sound):
    for sample in getSamples(sound):
        value = getSampleValue(sample)
        setSampleValue(sample, value * 0.5)

```

程序原理

- 该函数接受一个声音对象作为输入，在decreaseVolume函数内，输入声音将称为sound——不论它在命令区中叫什么名字。
 - 变量sample将代表输入声音中的每个样本。
 - 每次把新的样本赋给sample时，我们将取得样本的值并放入value中。
 - 然后，我们把value乘以0.5，把样本的值设为这个值，于是样本值变成了当前值的50%。
- 可以这样使用函数：

```

>>> f=pickAFile()
>>> print f
C:/ip-book/mediasources/louder-test.wav
>>> sound=makeSound(f)
>>> explore(sound)
>>> play(sound)
>>> decreaseVolume(sound)
>>> explore(sound)
>>> play(sound)

```

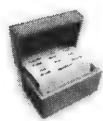
我们甚至可以再做一次，把音量进一步减小。

```
>>> decreaseVolume(sound)
>>> explore(sound)
>>> play(sound)
```

6.3.4 理解声音函数

函数处理图片时的工作原理（见3.4.1节）在这里同样适用于声音的处理。举例来说，我们可以把pickAFile和makeSound函数调用直接放进increaseVolume和decreaseVolume这样的函数中，但那意味着函数不再“做且只做一件事”。如果我们需要把很多段声音的音量提高或降低，那么就会觉得不断选取文件很烦人。

我们可以编写一个接受多个输入的函数。比如，下面是一个changeVolume菜谱。它接受一个factor（因子）。这个factor将与每个样本的值相乘。这个函数可以用来增加或减少振幅（从而增加或减小音量）。



程序56：基于给定的因子调整音量

```
def changeVolume(sound, factor):
    for sample in getSamples(sound):
        value = getSampleValue(sample)
        setSampleValue(sample, value * factor)
```

这份菜谱显然比increaseVolume和decreaseVolume更灵活。这是否表明它更好呢？对某些目标而言它当然更好（比如当你编写通用音频处理软件的时候），但对另一些目标而言，针对增加或减小音量分别提供独立且名字清晰的函数可能更好。别忘了软件是为人而编写的——要为阅读和使用软件的人编写可理解的软件。

我们把名字sound用了很多次。我们也用它命名过从命令区中读入的声音，还用它做过函数输入的占位符。这没有问题。名字可以在不同的上下文中拥有不同的含义。函数内部与命令区是不同的上下文。如果你在函数上下文中创建一个变量（比如程序56中的value），那么当你从函数退回命令区之后那个变量就不存在了。我们可以使用return从函数上下文返回值到命令区（或其他调用它的函数）。这一点后面会讲到更多。

6.4 声音规格化

仔细想一想，前面两个菜谱可以工作的事实会让你感到奇怪。我们可以把表示声音的数字都乘以一个数——而声音听起来还是一样的，只是更响一些。原因在于我们对声音的体验更多地取决于这些数字之间的关系，而不是这些数字本身。记住：声波的整体形状依赖于许多样本。一般来讲，把所有的样本乘以同样的因数只会影响我们对音量（强度）的感觉，而不会影响声音本身。（我们将在后面几节中编写程序修改声音本身。）

在人们对声音的处理中，常见的操作就是把它们变得尽可能的响亮。这称为声音的规格化（normalize）。这其实并不难，只是需要多用几个变量。以下是用语言描述的菜谱，我们就需要告诉计算机做这些事。

- 我们必须找出声音中的最大样本。如果它已经是最大值（32 767），那实际上我们已经没有办法在保证声音听起来一样的前提下增大音量。别忘了我们需要把所有样本都乘以同样的因数。

找出最大值的菜谱（算法）很容易——它是整个规格化菜谱的子菜谱。定义一个名字（比

如largest)并给它赋一个小值(0就可以)。然后检查所有的样本。如果找到比largest更大的,就修改largest使之持有这个更大的值。继续检查样本,但现在是与新的最大值比较。最后,数组中的最大值就存在于变量largest中。

要实现这一算法,我们需要一种从两个值中确定较大值的方法,Python提供了一个名叫max的内置函数,它可以完成这一功能。

```
>>> print max(8,9)
9
>>> print max(3,4,5)
5
```

- 下一步,我们需要确定那个与所有样本相乘的值。我们想让最大的值变成32 767。于是,我们需要确定一个因数(multiplier),使得:

$$(multiplier)(largest) = 32\ 767$$

求出multiplier:

$multiplier = 32\ 767 / largest$ 。因数需要用一个小数(具有小数部分)表示,因此,我们要让Python觉得至少有一个操作数不是整数。这很容易——使用32 767.0即可。只需加上“.0”。

- 现在,遍历所有样本,就像increaseVolume那样,将因数乘到每个样本上。以下便是规格化声音的菜谱。



程序57: 将声音规格化成最大振幅

```
def normalize(sound):
    largest = 0
    for s in getSamples(sound):
        largest = max(largest, getSampleValue(s))
    multiplier = 32767.0 / largest
    print "Largest sample value in original sound was", largest
    print "Multiplier is", multiplier

    for s in getSamples(sound):
        louder = multiplier * getSampleValue(s)
        setSampleValue(s, louder)
```

关于这个程序,有以下几点需要注意:

- 程序中存在空行! Python并不关心空行。增加空行可以把较长的程序分隔成多个部分,从而有助于理解。
- 程序中有print语句! print语句很有用。第一,它能提供一些关于程序正在运行的反馈——对长时间运行的程序而言,这非常有用;第二,它们显示了程序找到的样本,这些内容很有趣;第三,它是极好的测试方法,也是一种调试程序的方法。设想一下,如果输出中显示的因数小于1.0,那么我们就知道这样的因数会减小音量,这时你应该怀疑一定是哪里错了。
- 菜谱中某些语句很长,导致它们在文本中被自动换行。一定要在同一行上输入它们! Python不允许在语句结束之前按回车(Enter)——要确保print语句都没有断行。

以下是程序运行的情况。

```
>>> f = "c:/ip-book/mediasources/test.wav"
>>> s = makeSound(f)
>>> explore(s)
>>> normalize(sound)
Largest sample value in original sound was 11702
Multiplier is 2.8001196376687747
>>> explore(s)
>>> play(sound)
```

是不是很高兴？最有趣的部分是直接听到声音比原来响亮得多，这个很难在书本上演示。

产生削波

之前我们谈到过削波（clipping），正常的声音波形因为样本容量的限制而被剪断的效应。产生削波的方法之一就是不断增大音量。另外一种方法是显式地强制削波。

如果只有最大和最小样本值，也就是说，所有正值都是最大样本值，所有负值都是最小样本值，那么结果会是什么状况？试一下下面的菜谱，特别要针对说话的声音。



程序58：将所有样本设成最大值

```
def onlyMaximize(sound):
    for sample in getSamples(sound):
        value = getSampleValue(sample)
        if value >= 0:
            setSampleValue(sample, 32767)
        if value < 0:
            setSampleValue(sample, -32768)
```

最终结果似乎真的很怪异（如图6.18所示）。回放这段声音，你会听到一些令人不舒服的噪声。那就是削波的效果。而令人惊奇的是：经这个函数处理之后，你依然可以分辨出声音中的单词。我们从噪声中解码单词的能力强大得让人难以置信。

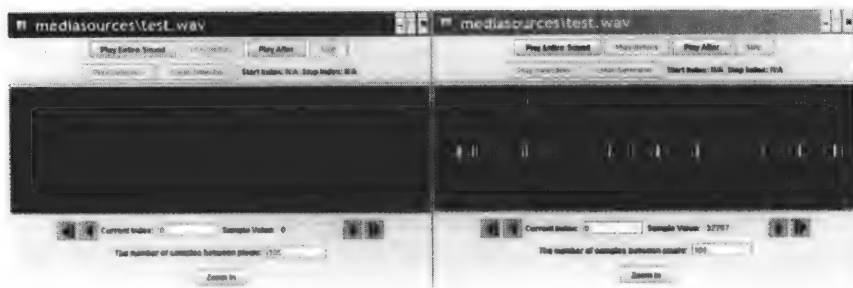


图6.18 原始声音和只有最大值的声音

编程摘要

本章讨论了以下几种数据（或对象）的编码。

声音	声音的编码，通常来自WAV文件
样本数组	样本对象的集合，每个样本用一个数字作下标（比如样本0，样本1）。samples[0]是第一个样本对象。你可以这样处理样本数组中的每个样本：for s in samples:
样本	大约在-32 000到32 000之间的一个值，表示录音时麦克风在给定瞬间产生的电压。瞬间的间隔一般是1/44 100秒（CD音质）或1/22 050秒（多数计算机上足够好的音质）。样本对象记录了自己所属的声音对象，因此，如果你改变了它的值，那么它“知道”去哪里修改相应样本

以下是本章使用或介绍过的函数：

<code>int</code>	返回输入值的整数部分
<code>max</code>	接受任意多个数字，返回其中最大的值

声音文件函数和程序片段

<code>pickAFile</code>	让用户选取一个文件并以字符串形式返回其完整路径名。没有输入
<code>makeSound</code>	接受文件名作为输入，读取文件内容并创建声音。返回新建的声音对象

声音对象函数和程序片段

<code>play</code>	播放输入的声音。没有返回值
<code>getLengh</code>	接受声音对象作为输入，返回声音中的样本数目
<code>getSamples</code>	接受声音对象作为输入，用一维数组的形式返回声音中的样本
<code>blockingPlay</code>	播放输入的声音并保证不会有其他声音同时播放。（试着以不同次序调用两个播放函数）
<code>playAtRate</code>	接受一个声音对象和一个速度（1.0表示正常速度，2.0比正常速度快一倍，0.5表示正常速度的一半）并以该速度播放声音。但播放时间是不变的（比如，两倍速播放时声音会播放两遍，从而占满所有时间）
<code>playAtRateDur</code>	接受声音对象、速度和以样本数目表示的持续播放时间
<code>writeSoundTo</code>	接受一个声音对象和一个文件名（字符串）并将声音作为WAV文件输出（如果想让操作系统正确对待输出的文件，要确保文件名以“.wav”结尾）
<code>getSamplingRate</code>	接受声音对象作为输入，返回一个数字，表示声音中每秒钟的采样数目

面向样本的函数和程序片段

<code>getSampleValueAt</code>	接受一个声音对象和一个（整数）下标，返回相应样本对象的值（-32 000~32 000）
<code>setSampleValueAt</code>	接受一个声音对象，一个下标和一个值（大致应该在-32 000~32 000），将给定声音中给定下标处的样本设置成给定的值
<code>getSampleObjectAt</code>	接受一个声音对象和一个（整数）下标，返回下标处的样本对象
<code>getSampleValue</code>	接受一个样本对象，返回它的值（-32 000~32 000）
<code>setSampleValue</code>	接受一个样本对象和一个值，将样本设成这个值
<code>getSound</code>	接受一个样本对象，返回所属的声音对象

习题

6.1 名词解释：

1. 削波
2. 规格化
3. 振幅
4. 频率
5. 疏部

6.2 分别写出数字-9、4和-27的二进制补码表示。

6.3 打开Sonogram视图，对着麦克风说几个词。不同的词会有不同的模式吗？所有的“噢”

模式都一样吗？所有的“啊”呢？如果换个人来说会有什么不同吗？模式还是一样的吗？

- 6.4 找一些不同的乐器，然后在MediaTools应用程序的声音编辑器中打开声谱视图，用这些乐器分别演奏同一个音符。所有“C”的图谱都一样吗？基于MediaTools的视觉化效果，你能看出这些不同乐器的区别吗？
- 6.5 试着用MediaTools声音编辑器中的琴键来“演奏”多个WAV文件。哪种录音最适合这样的“演奏”？
- 6.6 增大音量的菜谱（程序54）接受声音对象作为输入。编写一个increaseVolumeNamed函数，接受文件名作为输入，然后播放更响亮的声音。
- 6.7 编写一个函数，把声音样本中所有正值的音量增加，所有负值的音量减小。你还能听懂声音中的话吗？
- 6.8 编写一个函数找出声音样本中的最小值并打印出来。
- 6.9 编写一个函数统计声音中值为0的样本数目并把总数打印出来。
- 6.10 编写一个函数，把声音样本中大于0的值都置为最大正值（32 767）。
- 6.11 重写增大音量的函数（程序54），让它接受两个输入：要增大音量的声音和存储新的响亮声音的文件名。然后增大音量并将声音输出到名字所指定的文件。你还可以尝试把源声音也用文件名输入，而不是声音对象，这样两个输入参数便都是文件名了。
- 6.12 重写增大音量的函数（程序54），让它接受两个输入：要增加音量的声音和一个因数。用因数指定声音样本的振幅增加多少。能用这个函数同时完成音量的增大或减小吗？示范一下增大和减小音量分别应执行什么命令。
- 6.13 在6.3.2节中，我们一步步分析了程序54的工作原理。以同样的方式画图演示程序55的原理。
- 6.14 如果音量增加得太多会有什么结果？创建一个声音对象，增加它的音量，然后再增加一次，再增加一次，考察一下会是什么结果？它会一直变得更响吗？还是会发生其他事情？你能解释原因吗？
- 6.15 试着在声音中置入一些特定值。把一段声音中间的几百个样本值设成32 767，结果会怎样？设成-32 768，或者设成一串0，又会怎样？
- 6.16 试着给所有样本加上一个值，而不是乘一个因数（比如2或0.5）。把每个样本加上100，声音会怎样？加1 000呢？

深入学习

心理声学 and 计算机音乐方面有很多内容精彩纷呈的书籍。从理解的难度上讲，Mark最喜欢的一本是Dodge和Jerse写的《Computer Music: Synthesis, Composition and Performance》[11]。计算机音乐的“圣经”是Curtis Road的大部头《The Computer Music Tutorial》[35]。

使用MediaTools应用时，你实际是在使用一种称为Squeak的语言，最早它主要由Alan Kay、Dan Ingalls、Ted Kaehler、John Maloney和Scott Wallace开发[25]。Squeak现在是优秀的开源[⊖]跨平台媒体工具。有一本书[19]介绍了Squeak语言，包括它的声音处理能力，另外一本讲Squeak的书[20]中有关于Siren的一章，Siren是Squeak的一个变种，由Stephen Pope设计开发，专门用于计算机音乐的研究和创作。

⊖ <http://www.squeak.org>。

修改一段样本区域

本章学习目标

本章媒体学习目标：

- 剪接声音创建声音组合。
- 声音倒置。
- 声音镜像。

本章计算机科学学习目标：

- 用下标变量迭代数组中的一段范围。
- 在程序中使用注释并理解为什么要使用。
- 找出一些可用于多种媒体的算法。

7.1 用不同方法处理不同声音片段

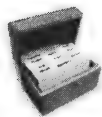
上一章我们描述了对声音作整体处理的几种方法，然而，为获得真正有趣的效果，我们还可以把声音截断并以不同的方式处理各个片段：某些语音这样处理，其他声音那样处理。如何做到呢？我们需要遍历一部分样本而不是全部样本的能力。事实上这不难做，但我们需要用稍微不同的方法来处理样本（比如需使用下标），而且需要以稍微不同的方式使用for循环。

还记得吗？每个样本都关联着一个数字，我们可以用`getSampleValueAt`（以声音和下标数字作为输入）来获得各个样本，用`setSampleValueAt`（以声音、下标和新值作为输入）来设置一个样本。正因为有这些方法，我们即使不使用`getSamples`和样本对象也同样能处理样本。但我们还是不愿编写这样的代码：

```
setSampleValueAt(sound,1,12)
setSampleValueAt(sound,2,28) ...
```

成千上万的样本，我们写不起！

之前用`getSamples`做的事情，使用`range`也都能做，只不过需要直接引用下标数字。下面是用`range`重写的程序54。



程序59：使用`range`增加输入声音的音量

```
def increaseVolumeByRange(sound):
    for sampleIndex in range(0,getLength(sound)):
        value = getSampleValueAt(sound,sampleIndex)
        setSampleValueAt(sound,sampleIndex,value * 2)
```

尝试一下——你会发现它与之前的程序功能完全相同。

但现在我们可以对声音做一些真正有趣的操作，因为我们可以控制哪些样本要被处理。下一份菜谱增大了声音前半部分的音量，减小了后半部分的音量。尝试一下，看看自己能否跟踪它的工作方式。



程序60：先增大音量，后减小音量

```
def increaseAndDecrease(sound):
    for sampleIndex in range(0, getLength(sound)/2):
        value = getSampleValueAt(sound, sampleIndex)
        setSampleValueAt(sound, sampleIndex, value * 2)
    for sampleIndex in range(getLength(sound)/2, getLength(sound)):
        value = getSampleValueAt(sound, sampleIndex)
        setSampleValueAt(sound, sampleIndex, value * 0.2)
```

程序原理

increaseAndDecrease函数中有两个循环，每个循环处理一半声音。

- 第一个循环处理声音当中从0到中点的样本。把这些样本都乘以2，从而振幅增加一倍。
- 第二个循环从中点开始一直处理到声音结束。这里我们把每个样本乘以0.2，从而它们的音量减小了80%。

引用数组元素的另一种写法

有必要指出的是：许多语言都使用方括号（[]）作为访问数组元素的标准写法。Python也是这样。对任何数组而言，array[index]都返回数组中的第index个元素。方括号中的数字永远是一个索引变量，但考虑到数学上引用 a 中第 i 个元素的写法，比如 a_i ，有时也称它为下标。

我们使用样本来说明一下。

```
>>> file = "C:/ip-book/mediasources/a.wav"
>>> sound = makeSound(file)
>>> samples = getSamples(sound)
>>> print samples[0]
Sample at 0 with value -90
>>> print samples[1]
Sample at 1 with value -113
>>> print getLength(sound)
9508
>>> samples[9507]
Sample at 9507 with value -147
>>> samples[9508]
The error was: 9508
Sequence index out of range.
The index you're using goes beyond the size of that
data (too low or high).
For instance, maybe you tried to access OurArray[10]
and OurArray only has 5 elements in it.
```

数组中第一个元素的下标为0。最后一个元素的下标为数组长度减1。

我们用range构造一个数组[⊖]，然后用同样的方法引用它。

```
>>> myArray = range(0,100)
>>> print myArray[0]
0
>>> print myArray[1]
1
>>> print myArray[99]
```

⊖ 技术上讲是个序列（sequence），序列也可以像数组那样索引，但不具备数组的全部特征。


```

99
>>> mySecondArray = range(0,100,2)
>>> print mySecondArray[35]
70

```

`range(0, 100)`创建了一个包含100个元素的数组。第一个元素下标为0，最后一个元素下标为99。数组持有0到99之间的所有值。记住：使用`range(begin, end)`来指定一段范围时，数字`begin`包含在返回的数组中，但`end`不包含在内。

7.2 剪接声音

剪接（splice）这个术语可以追溯到使用磁带录音的年代——调整磁带上声音内容的次序，也就是把磁带剪成小段，然后再以适当的次序粘起来。这就是“剪接”。一切都数字化之后，剪接就比以前容易多了。

要剪接声音，我们只需将数组中的元素到处复制。为了完成这种功能，在同一个数组内部复制会麻烦一些，使用两个（或更多）数组最容易。如果把表示某人说单词“the”的所有样本都复制到声音开头（从下标0开始），那就可以让声音以单词“the”开始。剪接能使你创建各种各样的声音：演讲、无意义的发声，甚至艺术品。

当声音存放于不同文件中时，剪接操作最简单。你只需要按次序将每一段声音复制到目标声音即可。下面的菜谱创建了一句话的开头：“Guzdial is ...”。（欢迎读者把这句话补全。）



程序61：将单词合并到一句话中

```

def merge():
    guzdialSound = makeSound(getMediaPath("guzdial.wav"))
    isSound = makeSound(getMediaPath("is.wav"))
    target = makeSound(getMediaPath("sec3silence.wav"))
    index = 0
    # 复制 "Guzdial"
    for source in range(0, getLength(guzdialSound)):
        value = getSampleValueAt(guzdialSound, source)
        setSampleValueAt(target, index, value)
        index = index + 1
    # 复制0.1秒的停顿（静音）(0)
    for source in range(0, int(0.1*getSamplingRate(target))):
        setSampleValueAt(target, index, 0)
        index = index + 1
    # 复制 "i秒"
    for source in range(0, getLength(isSound)):
        value = getSampleValueAt(isSound, source)
        setSampleValueAt(target, index, value)
        index = index + 1
    normalize(target)
    play(target)
    return target

```

程序原理

函数`merge`中有三个循环，每个循环都复制一小段声音到目标声音中——片段要么是个单词，要么是单词之间的静音。

- 函数首先创建单词“Guzdial”（`guzdialSound`）和单词`is`（`isSound`）的声音对象，以及静音的目标（`target`）。

- 注意，我们在循环开始之前让（针对目标的）`index`等于0。之后每次循环都递增它，但我们再也没有把它设为某个具体的值。这是因为`index`永远是目标声音中下一个空样本的下标。由于每次循环都紧跟着上一次循环，每次我们只需要把样本附加到目标末尾即可。
- 在第一组循环中，我们从`guzdialSound`复制各个样本到`target`中。我们让下标`source`从0开始一直到`guzdialSound`的长度。我们从`guzdialSound`中获得下标`source`处的样本值，然后把`target`声音中`index`处的样本值设为我们从`guzdialSound`中取得的值，最后递增`index`让它指向下一个空样本。
- 第二组循环中，我们增加了0.1秒的静音。由于`getSamplingRate(target)`提供了`target`中每秒钟的样本数目，乘以0.1就能给出0.1秒内的样本数目。这里没有使用任何源声——仅仅是把第`index`个样本设成0（静音），然后`index`加1。
- 最后，我们复制了`isSound`中的所有样本，就像第一组循环中复制`guzdialSound`时那样。
- 我们把声音`normalize`（规格化），使它更响。这意味着`normalize`函数必须和`merge`一起放在程序区中，尽管这里没有显示它。我们可以`play`（播放）并`return`（返回）声音。

之所以要用`return target`返回目标声音，是因为声音是在函数内部创建的，而不是从函数输入中传入的。如果我们不返回`merge`内部创建的声音，那么它将随着`merge`函数上下文（作用域）的终止而消失。返回这个结果，另一个函数就可以使用它。

除了用3秒静音作为目标之外，我们也可以使用`makeEmptySound(lengthInSamples)`构造一段指定长度的“空白”声音。比如，我们可以创建一段声音，它的长度正好等于要复制的内容，像下面这样：

```
guzLen = getLength(guzdialSound)
silenceLen = int(0.1 * 22050)
isLen = getLength(isSound)
target = makeEmptySound(guzLen + silenceLen + isLen)
```

新建声音的默认采样率是每秒钟22 050次。所以，想获得单词之间1/10秒的静音，静音的样本长度应该是 $0.1 \times 22\,050$ 。必须使用`int(0.1 * 22050)`把结果转换成一个整数，还可以用`makeEmptySound(length, samplingRate)`来创建新的静音。

更常见的一类剪接是这样的：一些单词位于已有声音的中间，需把它们从声音中摘录出来。在这种剪接操作中，首先要确定分隔各段内容的下标，也就是找出哪些片段是你感兴趣的内容。使用JES声音工具，这一点都不难。

- 在JES声音工具中打开WAV文件。可通过创建并查看声音来打开。
- 滑动窗口并移动光标（在图形中拖拽），直到觉得光标已处在关心的声音前面或后面。
- 使用声音工具中的按钮，播放光标前后的声音来检验你的定位是否正确。

就是用这样的过程，Mark找到了PREAMBLE10.WAV中前面几个单词的结束点（如图7.1所示）。他假定第一个单词从下标0开始。

编写从一个数组向另一个数组复制内容的循环需要一点小技巧。你需要同时想着两个下标：源数组中你走到哪儿了，目标数组中你又走到哪儿了。这是用两个不同的变量跟踪两个不同的下标，但它们都以同样的方式递增。

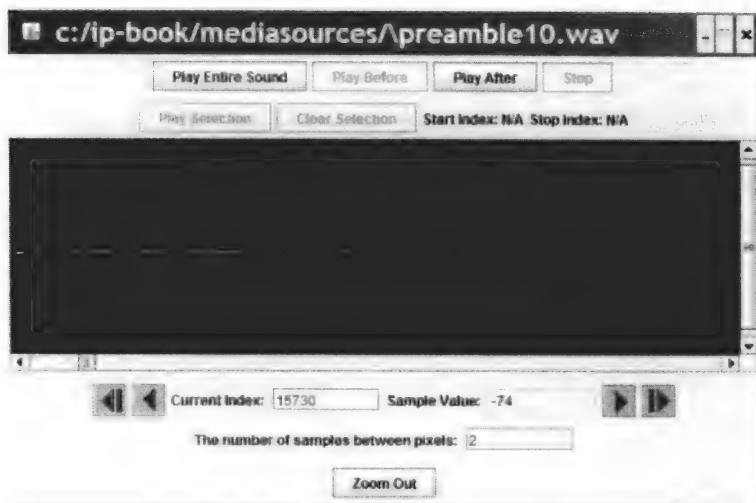


图7.1 通过查看声音找到单词之间的静音

将采用的方法（另一份子菜谱）是用一个下标变量指向目标数组中的正确入口，再用一个循环让第二个下标变量在源数组中沿各个项目移动，并且（非常重要！）每次复制一个元素之后都移动目标下标变量。我们就用这种方法来保持两个下标变量的同步。

我们通过加1来移动目标下标。这很简单，我们只是让Python执行 `targetIndex = targetIndex + 1`。还记得吗？这会使 `targetIndex` 的值变成它的当前值加1，从而移动（递增）了目标下标。我们在循环命令中修改源下标，如果再把上面的语句放进它的循环体中，就可以实现两个下标的同步移动。

子菜谱的通用形式为：

```
targetIndex = Where-the-incoming-sound-should-start
for sourceIndex in range(startingPoint, endingPoint)
    setSampleValueAt(target, targetIndex,
        getSampleValueAt(source, sourceIndex))
    targetIndex = targetIndex + 1
```

下面的菜谱把宪法前言中的 “We the people of the United States” 改成了 “We the united people of the United States”。



程序62：剪接朗读的宪法前言，使其中出现 “United People”

在把程序输入到你的计算机中之前别忘了修改文件变量名。

```
# 声音剪接
# 使用朗读宪法前言的录音制作 "We the united people"
def splicePreamble():
    file = "C:/ip-book/mediasources/preamble10.wav"
    source = makeSound(file)
```

```

# target将成为拼接后新的声音
target = makeSound(file)

# 在新的声音中, targetIndex刚好从"We the"结束的地方开始
targetIndex = 17408
# 单词"United"在声音中的位置
for sourceIndex in range(33414, 40052):
    value = getSampleValueAt(source, sourceIndex)
    setSampleValueAt(target, targetIndex, value)
    targetIndex = targetIndex + 1

# 单词"People"在声音中的位置
for sourceIndex in range(17408, 26726):
    value = getSampleValueAt(source, sourceIndex)
    setSampleValueAt(target, targetIndex, value)
    targetIndex = targetIndex + 1

# 在这之后, 附上一些静音区域
for index in range(0, 1000):
    setSampleValueAt(target, targetIndex, 0)
    targetIndex = targetIndex + 1

# 让我们听一下声音并返回结果
play(target)
return target

```

函数可以这样使用:

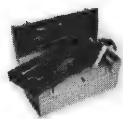
```
>>> newSound = splicePreamble()
```

程序原理

这份菜谱做了很多事情, 我们把它慢慢过一遍。

注意程序中有好多带“#”的行。这一符号表示这一行中它后面的内容都是写给程序员看的注解, Python应当忽略它们。这称为注释 (comment)。

实践技巧: 注释有益



注释是向别人 (也向自己) 解释程序做了什么的极好方法! 事实常常是: 你很难记住程序的所有细节, 因此, 留下一些注解来解释自己做了什么非常有用。有可能你会再次摆弄这个程序, 也可能其他人需要理解它。

splicePreamble函数没有参数。如果能编写一个随意剪接各种声音的函数, 就像我们把增大音量和规格化声音做成通用函数那样, 当然再好不过了。但这里如何做到呢? 如何能把所有起点和终点通用化? 看来, 至少在开始阶段, 编写只处理特定剪接任务的菜谱更容易些。

这里我们看到了之前构建的三个复制样本的循环。实际上只有两个。第一个把单词“united”复制到位。第二个把单词“people”复制到位。“等一下,” 你可能在想, “单词‘people’已经在声音中了!” 没错, 但把“united”复制进来时, 我们覆盖了一部分“people”, 因此需要再复制一遍。

菜谱的最后我们返回了目标声音target。target是函数内部创建的, 而不是参数传进来

的。如果不返回它，就无法再次引用它。返回它，我们就有机会给它起个名字，并在函数结束之后播放它（甚至进一步处理它）。

下面是形式更简单的程序。试着运行一下，听一听产生的结果。

```
def spliceSimpler():
    file = "C:/ip-book/mediasources/preamble10.wav"
    source = makeSound(file)
    # target将成为拼接后新的声音
    target = makeSound(file)
    # 在新的声音中，targetIndex刚好从"We the"结束的地方开始
    targetIndex = 17408
    # 单词"United"在声音中的位置
    for sourceIndex in range(33414, 40052):
        value = getSampleValueAt(source, sourceIndex)
        setSampleValueAt(target, targetIndex, value)
        targetIndex = targetIndex + 1
    # 让我们听一下声音并返回结果
    play(target)
    return target
```

我们来看看能否从数学上描述程序中发生的事情。回忆一下表7.1。我们从样本坐标为17 408的地方开始插入样本。单词“united”包含6638（40 052 - 33 414）个样本。（给读者留一道习题：若以时间计算那是多少秒呢？）这意味着我们要把下标17 408到24 046（17 408 + 6638）之间的样本写入目标。从表格中，我们知道单词“people”结束的位置在下标26 726处。如果单词“people”包含的样本超过2680（26 726 - 24 046）个，那它必然早于24 046开始，而我们插入的“united”将会踩到它的一部分。如果单词“united”有6000多个样本，我们怀疑单词“people”是不到2000的。这是它听起来有些细碎的原因。为什么“of”的位置没有问题？麦克风一定在那里停顿了一下。再次查表7.1，你会看到单词“of”的样本在下标32 131处结束，而它之前一个单词结束在26 726处。“of”不需要5405（32 131 - 26 726）个样本，这就是原来的菜谱可以正常工作的原因。

在程序62中，第三个循环看上去同样是复制样本的循环，但它实际上只是置入一些0值。0值样本表示静音。置入一些0值会产生停顿，使声音听起来更舒服。（后面有一个练习让你去除这些0值，试试能听出什么效果。）

图7.2把原来的PREAMBLE10.WAV文件显示在左边的声音编辑器中，剪接过的新文件（使用writeSoundTo保存的）显示在右边的声音编辑器中。两条线之间是剪接的部分，其余的声音都是一样的。



图7.2 比较原始声音（左）和剪接过的声音（右）

7.3 通用剪辑和复制

前面的函数看起来有些复杂。怎样使它简单一些呢？如果有这样一种通用的剪辑（clip）函数就好了：它接受一个声音对象、一个起点坐标和一个终点坐标，返回只包含原始声音中坐标指定部分的声音剪辑。那样一来，制作一段只包含“United”的声音剪辑就很容易了。



程序63：制作声音剪辑

```
def clip(source, start, end):
    target = makeEmptySound(end - start)
    targetIndex = 0
    for sourceIndex in range(start, end):
        sourceValue = getSampleValueAt(source, sourceIndex)
        setSampleValueAt(target, targetIndex, sourceValue)
        targetIndex = targetIndex + 1
    return target
```

现在我们可以通过下面的代码创建一段只包含单词“United”的录音剪辑。

```
>>> preamble = makeSound(getMediaPath("preamble10.wav"))
>>> explore(preamble)
>>> united = clip(preamble, 33414, 40052)
>>> explore(united)
```

同样，如果能有这样一种通用的复制方法也不错：接受源声音和目标声音，把源声音的内容复制到目标声音中由输入参数指定的起始位置。



程序64：通用复制

```
def copy(source, target, start):
    targetIndex = start
    for sourceIndex in range(0, getLength(source)):
        sourceValue = getSampleValueAt(source, sourceIndex)
        setSampleValueAt(target, targetIndex, sourceValue)
        targetIndex = targetIndex + 1
```

现在我们可以使用新的函数再次把单词“United”插入到朗读宪法前言的录音中。



程序65：使用通用剪辑和复制

```
def createNewPreamble():
    file = "C:/ip-book/mediasources/preamble10.wav"
    preamble = makeSound(file)
    united = clip(preamble, 33414, 40052)
    start = clip(preamble, 0, 17407)
    end = clip(preamble, 17408, 55510)
    len = getLength(start) + getLength(united) + getLength(end)
    newPre = makeEmptySound(len)
    copy(start, newPre, 0)
    copy(united, newPre, 17407)
    copy(end, newPre, getLength(start) + getLength(united))
    return newPre
```

注意这个函数是如何调用新的通用clip和copy函数的。你可以把这三个函数放进同一文

件中。但如果能有一个包含通用声音函数的文件，其他文件可以直接使用其中的函数而不需要把所有函数放进同一文件中，那就更好了。

可以通过从其他文件中导入（import）函数来实现这一目的。你需要在包含通用声音函数的文件中的第一行添加`from media import *`，从而可以使用JES提供的像`getMediaPath`或`getRed`这样的媒体函数。JES会自动帮你导入媒体函数，但现在我们不再通过JES加载通用文件，因此必须显式导入媒体函数。我们把包含通用声音函数的文件命名为`MYSOUND.PY`，这样它不会与JES中使用名字`Sound`的其他东西冲突。

```
from media import *

def clip(source, start, end):
    target = makeEmptySound(end - start)
    targetIndex = 0
    for sourceIndex in range(start, end):
        sourceValue = getSampleValueAt(source, sourceIndex)
        setSampleValueAt(target, targetIndex, sourceValue)
        targetIndex = targetIndex + 1
    return target

def copy(source, target, start):
    targetIndex = start
    for sourceIndex in range(0, getLength(source)):
        sourceValue = getSampleValueAt(source, sourceIndex)
        setSampleValueAt(target, targetIndex, sourceValue)
        targetIndex = targetIndex + 1
```

为使用新的通用声音函数，必须首先使用`setLibPath(path)`。这个函数告诉Python到哪里寻找导入的文件。`path`是包含通用函数的文件所在目录的完整路径名。

```
>>> setLibPath("c:/ip-book/programs/")
```

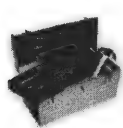
为了在另一个文件中使用通用声音函数`copy`和`clip`，在文件的第一行加入了`from mySound import *`。使用“`from mySound import *`”就好像把通用函数复制到了`createNewPreamble`所在的文件中，但实际效果比复制函数要好得多。如果我们把通用函数复制到很多文件中，然后修改通用函数，那么就必须到每一个复制它们的文件中分别修改它们。然而，如果通用函数是导入的，那么以后若修改了通用函数文件，其他地方使用的就是改进之后的函数了。

```
from mySound import *

def createNewPreamble():
    file = "C:/ip-book/mediasources/preamble10.wav"
    preamble = makeSound(file)
    united = clip(preamble, 33414, 40052)
    start = clip(preamble, 0, 17407)
    end = clip(preamble, 17408, 55510)
    len = getLength(start) + getLength(united) + getLength(end)
    newPre = makeEmptySound(len)
    copy(start, newPre, 0)
    copy(united, newPre, 17407)
```

```
copy(end,newPre, getLength(start) + getLength(united))
return newPre
```

通用函数也是一种抽象，就像在图片处理中用过的通用copy函数一样。创建一个包含通用函数的文件用于函数导入，其他人（也包括我们自己，因为自己也会忘）就可以直接使用这些抽象而不需要理解这些函数的实现细节，就像使用getRed和getMediaPath而无须了解其工作细节一样。



实践技巧：通用函数目录

如果把所有包含通用函数的文件都放在同一个目录中，那么，只需要调用一次setLibPath就能用import访问所有通用函数文件。你可能有分别用于声音、图片、电影等媒体的通用函数文件。

7.4 声音倒置

在剪接声音的例子中，我们原封不动地复制源声音中的单词样本。其实我们不一定非得以同样的次序复制。我们可以把单词倒置——或者使之更快、更慢、更响亮或更柔合。举个例子：以下就是个倒置声音的菜谱（如图7.3所示）。



程序66：逆序播放给定的声音（声音倒置）

```
def reverse(source):
    target = makeEmptySound(getLength(source))
    sourceIndex = getLength(source)-1
    for targetIndex in range(0,getLength(target)):
        sourceValue = getSampleValueAt(source,sourceIndex)
        setSampleValueAt(target,targetIndex,sourceValue)
        sourceIndex = sourceIndex - 1
    return target
```

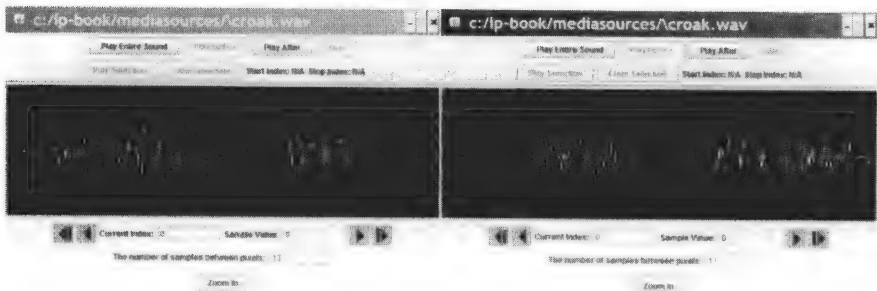


图7.3 比较原始声音（左）和倒置后的声音（右）

可以在命令区输入以下代码来尝试前面的程序：

```
>>> croak = makeSound(getMediaPath("croak.wav"))
>>> explore(croak)
>>> revCroak = reverse(croak)
>>> explore(revCroak)
```

程序原理

这份菜谱使用了我们见过的复制数组元素的子菜谱的一个变种。

- 首先，它创建了一个与source等长的空白声音target。
- 它让sourceIndex从数组的尾部（长度减1）而不是从前边开始。
- targetIndex从0到长度减1向后移动，菜谱在每一步：
 - 获得source中sourceIndex处的样本值，
 - 将这个值复制到target中的targetIndex处，并且，
 - 把sourceIndex减1，于是sourceIndex从数组尾部向着数组开头反向移动。

7.5 镜像

一旦学会了如何正向、反向地播放声音，镜像（mirror）一段声音的过程就跟镜像一幅图片完全一样了（如图7.4所示）。将下面的程序与程序20比较一下。你是否同意，尽管它们处理的是不同媒体，但它们是同一个算法吗？



程序67：从前向后镜像声音

```
def mirrorSound(sound):
    len = getLength(sound)
    mirrorpoint=len/2
    for index in range(0,mirrorpoint):
        left = getSampleObjectAt(sound,index)
        right = getSampleObjectAt(sound,len-index-1)
        value = getSampleValue(left)
        setSampleValue(right,value)
```

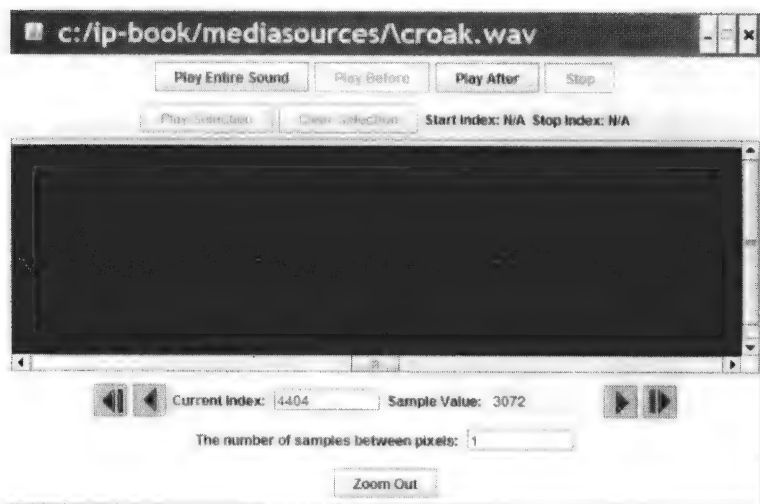


图7.4 从前（左）向后（后）镜像一段声音

编程摘要

本章讨论了以下几种数据（或对象）的编码。

声音	声音的编码，通常来自WAV文件
样本数组	样本对象的集合，每个样本用一个数字作为下标（比如样本0、样本1）。samples[0]是第一个样本对象。你可以这样处理样本数组中的各个样本：for s in samples:

(续)

样本 大约在-32 000到32 000之间的一个值，表示录音时麦克风在给定瞬间产生的电压。瞬间的间隔一般是1/44 100秒（CD音质）或1/22 050秒（多数计算机上足够好的音质）。样本对象记录了自身所属的声音对象，因此，如果改变了它的值，那么它“知道”去哪里修改相应样本

以下是本章使用或介绍过的函数：

range	接受两个数字，返回一个整数数组，包括从第一个数字开始，到最后一个数字之前的所有整数
range	还可以接受三个数字参数，返回一个整数数组，包括从第一个数字开始，到第二个数字结束（但不包括），增量为第三个数字的全部整数

习题

- 7.1 重写程序60，为函数提供两个输入：声音对象和一个百分数，其中百分数指定前面增大音量的部分占声音总长度的比例，该比例之后的部分减小音量。
- 7.2 重写程序60，将第一秒声音规格化，然后缓慢降低音量，每一秒比前一秒多降1/5。（每秒钟包含多少样本呢？getSamplingRate可以返回给定声音中每秒钟的样本数目。）
- 7.3 重写程序60，让前半部分的音量线性递增，后半部分的音量线性递减至0。
- 7.4 在剪接声音的例子中（程序62），如果去掉加到目标中的那一小段静音，试试看会有什么结果？你能听出不同吗？
- 7.5 为程序62编写一个新版本，将“We the”复制到新的声音中，再把“united”复制进来，最后把“people”复制进来。确保单词之间插入2250个0值样本。返回新的声音。
- 7.6 写一个新的clip函数，接受一个声音对象、一个起点坐标和一个终点坐标，返回只包含原始声音一部分的新声音。新声音的长度应为： $(endIndex - startIndex + 1)$ 。
- 7.7 我们觉得，剪接的声音（程序62）说“We the united people”时，“united”一词应该强调一下——即更响亮一些。试着修改程序，使短语“united people”中的“united”变得最响亮（规格化）。
- 7.8 试用秒表记录本章各程序的执行时间。计时应从命令区中输入回车开始，到下一次提示符出现为止。执行时间与声音长度之间有什么关联吗？是线性关系吗？（即更长的声音处理时间也更长，更短的声音处理时间也更短。）还是别的什么关系？再横向比较一下各个菜谱。声音规格化是否比提高（或降低）振幅消耗的时间更多？这个时间差是不变的吗？相差多少？与声音的长度是否有关系呢？
- 7.9 做一段音频剪辑。长度至少为5分钟，至少包含两段不同的声音（即来自不同的文件）。复制其中的一段并用本章描述的任一方法（比如镜像、剪接、音量处理）修改之。最后将原来的两段声音以及修改后的声音拼接在一起形成完整的剪辑。
- 7.10 编撰一个没有人说过的句子：把其他声音中的单词组合成一段语法正确的新声音。编写一个audioSentence函数，基于单词产生句子。句子中至少要使用三个词。你可以使用网站上MEDIASOURCE文件夹下面提供的单词，也可以自己录制单词。确保单词之间有1/10秒的停顿。（提示1：记住0值样本会产生静音或停顿。提示2：记住采样率是每秒内的样本数目。通过这些，你应该能算出需要把多少样本设成0才能产生1/10秒的停顿。）确保使用getMediaPath访问媒体文件夹中的声音，这样只要用户首先执行setMediaPath，程序就能正常运行。

- 7.11 编写一个函数在声音的开头添加1秒钟的静音。它应该接受声音对象作为输入，创建新的空白目标声音，然后在目标声音中从第1秒钟之后开始复制原来的声音。
- 7.12 到网上查找一些倒置后能听到隐藏信息的歌曲。
- 7.13 编写一个函数，剪接单词和音乐。
- 7.14 编写一个函数把两段声音交错起来。首先第一段声音2秒钟，接着第二段声音2秒钟。然后第一段声音再2秒钟，第二段声音再2秒钟。这样一直进行，直到两段声音全部复制到目标声音中。
- 7.15 编写一个erasePart函数，把“thisisatest.wav”第2秒钟中的所有样本都置成0——实质上就是让第2秒钟静音。（提示：记住getSamplingRate(sound)能告诉你一段声音中每秒内的样本数目。）播放并返回这段局部被抹掉的声音。
- 7.16 你能编写一个函数找出单词之间的静音部分吗？这个函数应该返回什么？
- 7.17 编写一个函数，传入声音对象和起点、终点下标，为这段声音中下标指定的部分增加音量。
- 7.18 编写一个函数，传入声音对象和起点、终点下标，倒置这段声音中下标指定的部分。
- 7.19 编写一个mirrorBackToFront函数，将声音中的后半部分镜像到前半部分。
- 7.20 编写一个reverseSecondHalf函数，接受一段声音作为输入，仅倒置声音的后半部分并返回结果。举例来说，如果声音中说的是“MarkBark”，返回的声音应当说“MarkkraB”。

通过合并片段制作声音

本章学习目标

本章媒体学习目标：

- 混合声音使它在另一段声音中逐渐消失。
- 制造回声。
- 改变声音的频率（音高）。
- 通过组合基本音（正弦波）制作自然界中并不存在的声音。
- 针对不同目的选择不同的声音格式（如MIDI和MP3）。

本章计算机科学学习目标：

- 使用文件路径引用磁盘上不同位置的文件。
- 将混合（融合）解释为跨不同媒体的算法。
- 使用多个函数构造程序。

8.1 用加法组合声音

采用数字化技术创造原本并不存在的声音会带来很多乐趣。除了简单的到处复制样本值和使用乘法运算外，实际上还可以修改样本的值，或者把不同声波叠加起来，结果可以产生以前不曾存在过的声音。

从物理上讲，声音的叠加涉及波的抵消以及强化其他因素的问题。从数学上讲，它与矩阵有关。从计算机科学上讲，它是人们能想到的最简单的过程之一。假设你得到一段声音`source`，想把它加入`target`中，只需将下标相同的样本值加起来即可（如图8.1所示）。就是这样！

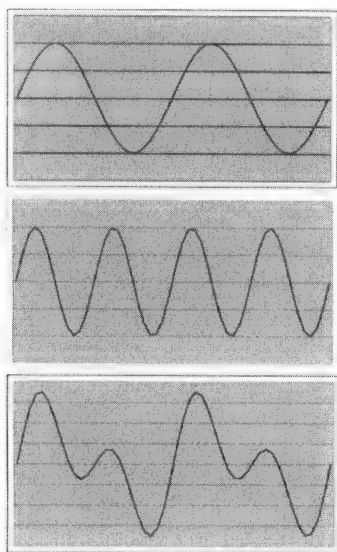


图8.1 顶上和中间的波叠加起来创建了底下的波

```

for sourceIndex in range(0, getLength(source)):
    targetValue = getSampleValueAt(target, sourceIndex)
    sourceValue = getSampleValueAt(source, sourceIndex)
    setSampleValueAt(target, targetIndex, sourceValue + targetValue)

```

setSampleValueAt函数接受一个声音对象、需要修改的样本下标，以及修改时使用的样本新值作为输入。它与setSampleValue类似，但无需首先从声音中取出样本对象。

为了使操作更加方便，我们将使用setMediaPath和getMediaPath函数。JES知道如何设定一个媒体目录（文件夹），然后引用目录中的媒体文件。这大大方便了媒体文件的引用——不需要写出完整的路径名。setMediaPath()函数允许你用文件选择器选择一个媒体目录。getMediaPath(baseName)接受一个基本文件名，将媒体目录添加到基本名之前并返回它。

```

>>> setMediaPath()
New media folder: C:/ip-book/mediasources/
>>> print getMediaPath("barbara.jpg")
C:/ip-book/mediasources/barbara.jpg
>>> print getMediaPath("sec1silence.wav")
C:/ip-book/mediasources/sec1silence.wav

```



常见bug：它不是文件，是字符串

getMediaPath返回一个看似文件名的东西，但这不足以表明相应的文件真正存在。你必须知道正确的基本名，只要你知道，在代码中用起来就很容易。但如果使用了不存在名字，那么你会得到一个指向不存在文件的路径，getMediaPath就会发出警告。

```

>>> print getMediaPath("blah-blah-blah")
Note: There is no file at C:/ip-
book/mediasources/blah-blah-blah
C:/ip-book/mediasources/blah-blah-blah

```

8.2 混合声音

在这个例子中，我们接受两段声音——某个人说的“啊！”和低音管发出的第4音阶的C——并混合（blend）它们。为此，先复制一部分“啊”，接着两段声音各50%，最后复制C。这与在混音板上将两段声音进行50%混合非常类似。与程序44中融合图片的方法也很类似。



程序68：混合两段声音

```

def blendSounds():
    bass = makeSound(getMediaPath("bassoon-c4.wav"))
    aah = makeSound(getMediaPath("aah.wav"))
    canvas = makeSound(getMediaPath("sec3silence.wav"))
    # 两段声音的样本数目都大于40KB
    for index in range(0, 20000):
        setSampleValueAt(canvas, index, getSampleValueAt(aah, index))
    for index in range(0, 20000):
        aahSample = getSampleValueAt(aah, index + 20000)
        bassSample = getSampleValueAt(bass, index)
        newSample = 0.5 * aahSample + 0.5 * bassSample
        setSampleValueAt(canvas, index + 20000, newSample)
    for index in range(20000, 40000):
        setSampleValueAt(canvas, index + 20000, getSampleValueAt(bass, index))
    play(canvas)
    return canvas

```

程序原理

与图片融合（程序44）类似，这个函数里也有针对每一段混合片段的循环。

- 首先，创建了用于混合的声音对象bass和aah以及存放混合结果的canvas。这些声音的长度都超过40 000个样本，但我们只会使用前面的40 000个作为示例。
- 第一个循环中，我们只从aah中取出20 000个样本复制到canvas中。注意我们没有为canvas使用单独的下标变量，而是将一个下标变量index同时用于两段声音。
- 下一个循环中，我们同时从aah和bass中复制20 000个样本并混合到canvas中。我们使用下标变量index同时索引三段声音——使用index本身访问bass，加上20 000访问aah和canvas（因为我们已经从aah中复制了20 000个样本到canvas中）。我们从aah和bass中各取一个样本，然后分别乘以0.5再把结果加起来。结果得到一个两者各占50%的样本。
- 最后，我们从bass中复制另外的20 000个样本。之后函数返回了结果声音（否则它会消失），这段声音听起来首先是“啊”，然后是两段声音各有一点，最后就只能听到低音管演奏的音符了。

8.3 制造回声

制造回声效果类似于我们在前一章见过的剪接声音的菜谱（程序62），但需要创建一段原先不存在的声音。我们通过波形的叠加来达到这一目的。这一次我们要做的是把样本加到delay个样本之后，但在相加之前先把它乘以0.6，从而使回声弱一些。



程序69：制作声音并加上一次回声

```
def echo(delay):
    f = pickAFile()
    s1 = makeSound(f)
    s2 = makeSound(f)
    for index in range(delay, getLength(s1)):
        # 将下标delay处的值设为原值加上延迟值乘以0.6
        echoSample = 0.6*getSampleValueAt(s2, index-delay)
        comboSample = getSampleValueAt(s1, index) + echoSample
        setSampleValueAt(s1, index, comboSample)
    play(s1)
    return s1
```

程序原理

echo函数接受回声与原声间的延迟量作为输入并返回带回声的声音。试着用不同的延迟量将程序运行一下。延迟值低的时候，回声听上去更像颤音（vibrato）。更高的延迟值（试一下10 000或20 000）才会提供真正的回声效果。

- 这个函数提示你选择一个用于制造回声的声音文件（如果想把回声函数用于其他目的，这却不是个好主意），然后创建声音的两份拷贝s1和s2。我们将在s1中创建带回声的声音，从s2中获取原先的、未掺加回声的样本用于创建回声。（你可以试试只用一个声音对象的情况，来获得有趣的分层叠加的回声。）
- index循环跳过了delay个样本，然后一直循环到声音结束。
- 回声要延后delay个样本，因此index-delay是我们需要的样本。我们把它乘以0.6，让音量更轻一些。
- 然后，我们把回声样本跟当前样本相加并保存在comboSample中，再把comboSample保存

在s1中下标为index的地方。

- 最后，我们play（播放）并return（返回）了声音。

8.3.1 制造多重回声

这个菜谱让你设置回声的数目。使用它，你可以产生一些令人惊奇的效果。



程序70：制造多重回声

```
def echoes(sndFile, delay, num):
    # 创建新的声音，针对输入的声音文件添加num次回声
    # 回声之间的间隔为delay
    s1 = makeSound(sndFile)
    ends1 = getLength(s1)
    ends2 = ends1 + (delay * num)
    # ends2是样本数目——必须转换成秒数
    s2 = makeEmptySound(1 + int(ends2 / getSamplingRate(s1)))

    echoAmplitude = 1.0
    for echoCount in range(1, num):
        # 每次减小为上次的60%
        echoAmplitude = echoAmplitude * 0.6
        for posns1 in range(0, ends1):
            posns2 = posns1 + (delay * echoCount)
            values1 = getSampleValueAt(s1, posns1) * echoAmplitude
            values2 = getSampleValueAt(s1, posns2)
            setSampleValueAt(s2, posns2, values1 + values2)
    play(s2)
    return s2
```

8.3.2 制作和弦

音乐和弦是指三个或更多的音符同时演奏时产生的和谐悦耳的声音。AC大和弦是C、E和G音符的组合。要制作和弦，可以简单地把相同下标的值加起来。MEDIASOURCES目录包含低音管分别演奏C、E和G音符的声音文件。



程序71：制作和弦

```
def createChord():
    c = makeSound(getMediaPath("bassoon-c4.wav"))
    e = makeSound(getMediaPath("bassoon-e4.wav"))
    g = makeSound(getMediaPath("bassoon-g4.wav"))
    chord = makeEmptySound(c.getLength())
    for index in range(0, c.getLength()):
        cValue = getSampleValueAt(c, index)
        eValue = getSampleValueAt(e, index)
        gValue = getSampleValueAt(g, index)
        total = cValue + eValue + gValue
        setSampleValueAt(chord, index, total)
    return chord
```

8.4 采样键盘工作原理

采样键盘 (sampling keyboard) 是使用录音 (比如钢琴、竖琴、小号的声音) 创作音乐的键盘, 它会按照期望的音调来演奏这些录音, 从而创作音乐。现代音乐和声音键盘 (以及合成器) 让音乐家们可以录下生活中的声音, 然后通过变换声音的原始频率, 把它们变成“乐器”声音。合成器是如何做到这一点的呢? 实际上并不复杂。有趣的部分在于: 它允许你把任何声音用做乐器。

采样键盘使用大量内存来保存大量不同的乐器在不同音高时的声音。当你按下键盘上的某个键时, 与你按下的音符 (在音高上) 最接近的录音将被选中, 然后精确地转换成你要求的那个音高。

下面的第一份菜谱每隔一个样本跳过一个, 以此创建新的声音。你没看错——经过这么长时间小心翼翼地所有样本同等对待, 这一次, 我们打算跳过一半! 在 `mediasources` 目录下, 看到一个名叫 `c4.wav` 的声音文件。这是钢琴第4音阶的C音符演奏1秒钟的录音。作为实验声音, 它很适合, 但任何声音也都可以。



程序72: 将声音的频率加倍

```
def double(source):
    len = getLength(source) / 2 + 1
    target = makeEmptySound(len)
    targetIndex = 0
    for sourceIndex in range(0, getLength(source), 2):
        sourceValue = getSampleValueAt(source, sourceIndex)
        setSampleValueAt(target, targetIndex, sourceValue)
        targetIndex = targetIndex + 1
    play(target)
    return target
```

下面是使用方法:

```
>>> file = pickAFile()
>>> print file
C:/ip-book/mediasources/c4.wav
>>> c4 = makeSound(file)
>>> play(c4)
>>> c4doubled=double(c4)
```

看上去这份菜谱使用了我们前面见过的复制数组的子菜谱, 但要注意: `range` 使用了三个参数——增量为2。

动手试一下! [⊖] 你会听到声音的频率真的增加了一倍。

这是怎么回事呢? 真的没那么复杂。你可以这么理解: 原先文件的频率实际上就是一段时间里经历的周期数目。如果每隔一个样本跳过一个, 那么新的声音周期数目仍跟原来一样, 占用的时间却少了一半。

现在我们来尝试另一种处理: 把每个样本取两次。结果会怎样呢?

为实现这一目标, 我们需使用Python的 `int` 函数返回其输入的整数部分。

⊖ 你现在是一边读书一边动手尝试, 对吧?


```
>>> print int(0.5)
0
>>> print int(1.5)
1
```

下面是把频率减半的菜谱。我们再次使用了复制数组的子菜谱，但把两个变量颠倒了一下。for循环沿着声音的长度移动targetIndex。显式递增的是sourceIndex——但每次只增加0.5。结果源声音中每个样本都取了两次。各次循环中sourceIndex的值依次为1、1.5、2、2.5……但由于使用了int，我们依次取用的样本下标实际是1、1、2、2……



程序73：频率减半

```
def halve(source):
    target = makeEmptySound(getLength(source) * 2)
    sourceIndex = 0
    for targetIndex in range(0, getLength(target)):
        value = getSampleValueAt(source, int(sourceIndex))
        setSampleValueAt(target, targetIndex, value)
        sourceIndex = sourceIndex + 0.5
    play(target)
    return target
```

程序原理

halve函数接受一段源声音作为输入，创建了长度两倍于源声音的目标声音。置sourceIndex为0（那是从source中复制样本的起始位置）。然后，我们用一个循环让targetIndex从0递增到target声音结尾。我们从source中下标为sourceIndex整数部分（int）的地方取得一个样本值，把目标声音中targetIndex处的样本值置为从source样本中取得的值。然后，让sourceIndex加0.5。这意味着sourceIndex将在各轮循环中依次成为：0、0.5、1、1.5、2、2.5……而这个序列的整数部分为：0、0、1、1、2、2……结果，我们把source声音中的每个样本取了两次。

思考一下我们编写的程序。如果把其中的0.5变成0.75、2或3，它还能正常工作吗？for循环需要做些改变，但在所有的情形中，基本思想是不变的。我们是在采样源数据来创建目标数据。使用采样率0.5会放缓声音并使频率减半。而大于1的采样率，则会加快声音并增大频率。

我们尝试用下面的程序将采样过程通用化。（注意，这个程序不会正常工作。）



程序74：变换声音的频率——错误的实现

```
def shift(source, factor):
    target = makeEmptySound(getLength(source))
    sourceIndex = 0

    for targetIndex in range(0, len):
        sourceValue = getSampleValueAt(source, sourceIndex)
        setSampleValueAt(target, targetIndex, sourceValue)
        sourceIndex = sourceIndex + factor

    play(target)
    return target
```

以下是使用方法：

```
>>> cF=getMediaPath("c4.wav")
>>> print cF
c:/ip-book/mediasources/c4.wav
>>> c4 = makeSound(testF)
>>> lowerC4=shift(c4,0.75)
```

看上去似乎正常。然而，如果采样factor超过1.0，结果会怎样呢？

```
>>> higherTest=shift(c4,1.5)
You are trying to access the sample at index: 67585,
but the last valid index is at 67584
The error was:
Inappropriate argument value (of correct type).
An error occurred attempting to pass an argument to a
function. Please check line 218 of C:\ip-book\
programs\mySound.py
```

这是为什么呢？发生什么事情了？如果在setSampleValueAt之前打印一下sourceIndex，就能看出是怎么回事了。你会看到sourceIndex变得比源声音的长度还大。当然，这也能说得通。每次循环把targetIndex加1，而sourceIndex的增量却大于1，结果必然会在目标声音到达末尾之前sourceIndex就已经越过了源声音的末尾。但如何防止这一点呢？

想要的效果是：如果sourceIndex超过源声音的长度，就把sourceIndex重置为0。Python中可以使用if语句实现仅在测试条件为真时才执行后序块中的代码。



程序75：变换声音的频率

```
def shift(source,factor):
    target = makeEmptySound(getLength(source))
    sourceIndex = 0

    for targetIndex in range(0, len):
        sourceValue = getSampleValueAt(source,sourceIndex)
        setSampleValueAt(target, targetIndex, sourceValue)
        sourceIndex = sourceIndex + factor
        if (sourceIndex >= sourceLen)
            sourceIndex = 0

    play(target)
    return target
```

实际上，我们可以设置一个因子来得到任何一种想要的频率。我们把这个因子称为采样间隔 (sampling interval)。对某种期望的频率 f_0 ，采样间隔应为：

$$\text{samplingInterval} = (\text{sizeOfSourceSound}) \frac{f_0}{\text{samplingRate}}$$

其中，sizeOfSourceSound为源声音的长度，samplingRate为采样率。

这就是键盘合成器的工作原理。它保存着钢琴、噪音、打击乐器等各种录音。通过以不同的采样间隔来采样这些声音，它就能把声音转换成想要的频率。

本节的最后一份菜谱首先以原始频率播放一段声音，然后分别以2倍频、3倍频、4倍频、5倍频来播放。



程序76：以多种频率播放声音

```
def playASequence(file):
    # 声音播放5次，频率依次递增
    for factor in range(1, 6):
        sound = makeSound(file)
        target = shift(sound, factor)
        blockingPlay(target)
```

采样算法

你应当意识到，频率减半的程序73与图片放大的程序35，有一些相似性。为了使频率减半，我们把下标变量每次增加0.5并用int()函数取其整数部分，从而每个样本取了两次。为了让图片更大，我们也把下标变量每次增加0.5并应用int()函数，于是每个像素取了两次。它们使用的是同样的算法（algorithm）——每个菜谱的基本流程都是一样的。图片的细节和声音的细节不是关键问题。关键问题就是每个菜谱的基本流程是一样的。

我们也见过其他跨不同媒体的算法。显然，增加红色和增加音量的函数（以及相应的“减少”版本）做了基本一致的事情。融合图片跟混合声音的方法也是一样的。我们取得（像素）颜色通道或（声音）样本并把它们加起来，用百分数来确定最终结果中各个输入所占的分量。只要比例的总和是100%，就能获得以恰当的百分比反映输入声音或图片的合理输出。

找出这类算法是有好处的，原因有很多。如果我们能从一般意义上理解算法（比如，它何时慢何时快，能用于什么不能用于什么，局限在哪里），那样就可以把学到的知识运用到具体的图片或声音实例上。对设计者来说，了解算法同样有帮助。设计新的程序时，脑子里想着算法，就可以在适用的场合使用它们。

把声音频率加倍或减半的时候，我们同时也缩减或倍增了声音的长度。你可能想要一段长度与原来完全一样的目标声音，而不必在更长的声音中清除多余的空间。可以用makeEmptySound达到这一目标。makeEmptySound(22050*10)返回采样率为22 050，长度为10秒的空白声音。

8.5 加法合成

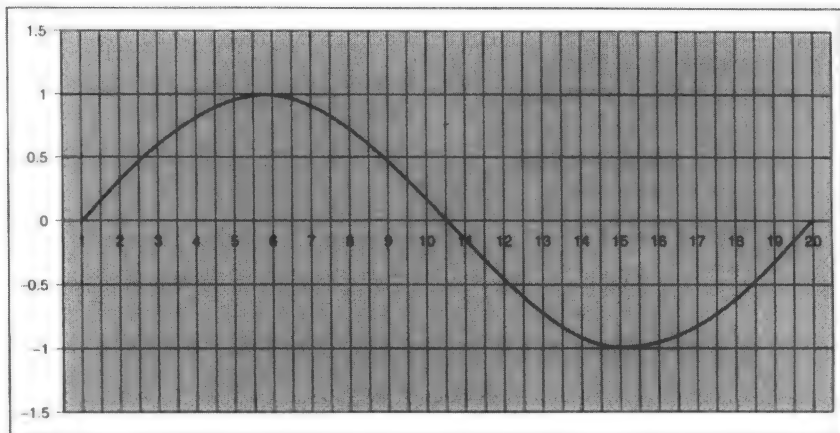
加法合成通过叠加正弦波来创建声音。之前我们看到，声音的叠加非常简单。使用加法合成，你可以手工控制波形，设置声波的频率，或创建从来不曾存在的“乐器”声音。

8.5.1 制作正弦波

考虑一下如何以给定的频率和振幅制造一组样本，从而产生声音。一种简单的方法是创建一段正弦波（sine wave）。举例来说，一声口哨几乎就是个完美的正弦波。巧妙之处在于如何以设想的频率创建正弦波。

如果从0到 2π 之间取一些值，计算每个值的正弦函数并绘制图形，就会得到一段正弦波。相信在很久之前的数学课上你已经学过：0和1之间有无穷多个数。计算机无法非常有效地处理“无穷大”，因此，我们只能从0~ 2π 之间取一些值。

为创建下面的图形，Mark用0~ 2π （约6.28）之间的值在一份电子表格中填了20行（随意的行数）。他让每一行上的数字比前一行多0.314（ $6.28/20$ ）。后面的一列计算了前一列各个数的正弦值，然后他基于这些值画出了图形。



如果想按给定的频率，比如440 Hz，来制作声音，那么必须在1/440秒内装下如上图所示的一个完整周期（每秒440个周期，也就是每个周期占用1/440秒，约等于0.002 27秒）。Mark画图时用了20个值。可以把它们称为20个样本。440 Hz的一个周期需要切成多少样本呢？这等于问0.002 27秒的时间里要经历多少个样本。我们知道采样率的概念——每秒钟的样本数目。假设采样率为每秒22 050个样本（我们的默认采样率），那么每个样本就是1/22 050秒，约等于0.000 045 3秒。那0.002 27秒内能装多少个样本呢？答案是0.002 27/0.000 045 3，大约50个。上面所做的推导可以用数学公式表示如下：

$$\text{interval} = 1 / \text{frequency}$$

$$\text{samplesPerCycle} = \frac{\text{interval}}{1 / \text{samplingRate}} = (\text{samplingRate})\text{interval}$$

其中，*frequency*是声波频率，*samplingRate*是采样率。

现在，用Python来表达这一公式。要得到给定频率（比如440 Hz）的波形，需要每秒钟有此种波形的440个周期。每个周期必须装进1/*frequency*秒的间隔（*interval*）中。每个周期间隔内需要产生的采样数目是采样率除以频率的商，或者说（1/*f*）乘以采样率。我们把它称为每周期的样本数目（*samplesPerCycle*）。

针对声音中的每个*sampleIndex*，进行：

- 计算分数*sampleIndex/samplesPerCycle*。
- 把这个分数乘以 2π 就是需要的弧度数。然后，计算(*sampleIndex/samplesPerCycle*)* 2π 的正弦值。
- 结果再乘以给定的振幅，用来设置*sampleIndex*处的样本。

为了构造声音，*mediasources*目录下放了一些静音文件。正弦波产生器将使用一秒钟的静音来构造一秒钟的正弦波。我们将提供一个振幅值*amplitude*作为函数的输入——它将是声音的最大振幅（由于正弦函数产生-1~1之间的数值，振幅的范围将在-*amplitude*到*amplitude*之间）。



程序77：以给定的频率和振幅产生正弦波

```
def sineWave(freq, amplitude):
```

```
    # 取得空白声音
```

```

mySound = getMediaPath('sec1silence.wav')
buildSin = makeSound(mySound)

# 设置声音常量
sr = getSamplingRate(buildSin)                # 采样率

interval = 1.0 / freq                          # 确保它是个浮点数
samplesPerCycle = interval * sr               # 每秒钟的采样数
maxCycle = 2 * pi

for pos in range(0, getLength(buildSin)):
    rawSample = sin((pos / samplesPerCycle) * maxCycle)
    sampleVal = int(amplitude * rawSample)
    setSampleValueAt(buildSin, pos, sampleVal)

return buildSin

```

注意，我们用1.0除以 $freq$ 来计算样本间隔。用1.0而不用1，可以确保结果是个浮点数而不是整数。如果Python看到所有的操作数都是整数，那么它计算的结果也会是整数，小数点之后的部分会被丢弃。至少使用一个浮点操作数（1.0），Python给出的结果也将是浮点数。

下面的命令以4 000作为振幅构建了一段880 Hz的正弦波。

```

>>> f880 = sineWave(880, 4000)
>>> play(f880)

```

8.5.2 把正弦波叠加起来

现在把正弦波叠加起来。本章一开始就讲过，这很容易：只需将相同下标的样本加起来即可。下面的函数将一段声音加到了另一段声音中。



程序78：叠加两段声音

```

def addSounds(sound1, sound2):
    for index in range(0, getLength(sound1)):
        s1Sample = getSampleValueAt(sound1, index)
        s2Sample = getSampleValueAt(sound2, index)
        setSampleValueAt(sound2, index, s1Sample + s2Sample)

```

把频率分别为440 Hz、880 Hz（440的两倍）和1 320 Hz（880 + 440）的三段声音加到一起，每段声音的振幅依次增加：2000、4000、8000。我们把它全部加到f440中并查看结果。最后，产生一段440 Hz的声音，从而可以把两段声音都听一下并做比较。

```

>>> f440 = sineWave(440, 2000)
>>> f880 = sineWave(880, 4000)
>>> f1320 = sineWave(1320, 8000)
>>> addSounds(f880, f440)
>>> addSounds(f1320, f440)
>>> play(f440)
>>> explore(f440)
>>> just440 = sineWave(440, 2000)
>>> play(just440)

```

```
>>> explore(just440)
```



常见bug：当心振幅相加超过32 767

叠加声音也叠加了它们的振幅。最大振幅分别为2 000、4 000和8 000的声音，加起来也不会超过32 767，不需要担心。但不要忘了上一章的例子，其中振幅太大时会是什么结果……

8.5.3 检查结果

我们如何知道程序确实给出了自己想要的结果呢？如果查看原来的f440和修改过的f440声音（后者实际上是三段声音的组合）你会注意到波形看起来很不一样（如图8.2所示）。这说明我们确实对声音做了某种修改……但到底是哪种修改呢？

可以用MediaTools中的声音工具来检验代码。首先，保存440 Hz的波just440和组合后的波。

```
>>> writeSoundTo(just440, "C:/ip-book/mediasources/just440.wav")
>>> writeSoundTo(f440, "C:/ip-book/mediasources/combined440.wav")
```

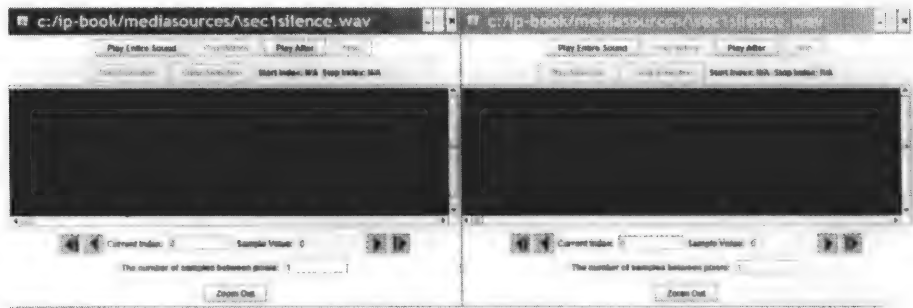


图8.2 未经处理的440 Hz信号（左）和440 + 880 + 1 320 Hz的信号（右）

真正能检验叠加合成的方法是使用快速傅立叶变换（Fast Fourier Transform, FFT）。使用MediaTools应用为每种信号产生FFT，将看到440 Hz的信号只有一个尖峰（如图8.3所示）。这是符合预期的——因为只有一种正弦波。再来看看合成波形的FFT，也是符合预期的。将看到

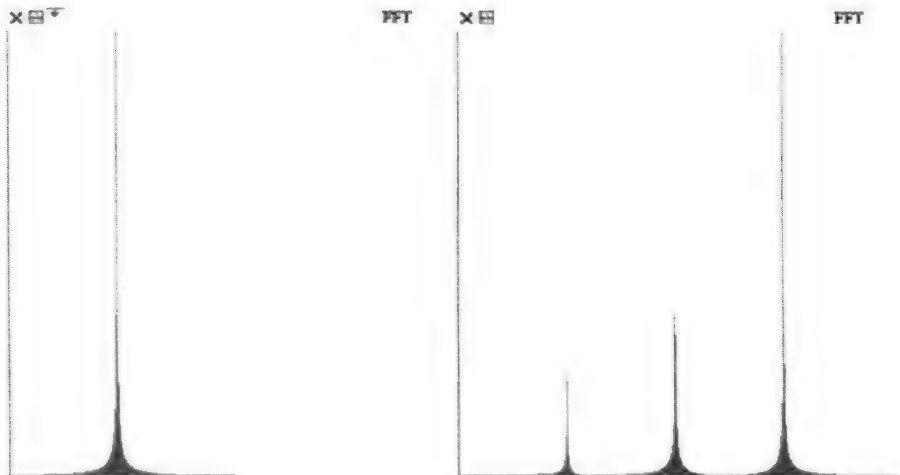


图8.3 440 Hz声音（左）和组合声音（右）的FFT

三个尖峰，且后面一个的高度总是前面一个的两倍。

8.5.4 方波

我们不仅可以叠加正弦波，还可以叠加方波 (square wave)。方波，顾名思义，就是方形的波，在+1~-1之间移动。其FFT结果看起来很不一样，声音也截然不同。实际上它能带来更加丰富的音质。

试着用下面的菜谱替换正弦波产生器，看看与你想象的是否一样。注意if语句的使用，它负责每次经过半个周期时在正信号和负信号之间切换。



程序79：针对给定频率和振幅的方波产生器

```
def squareWave(freq, amplitude):

    # 取得一段空白声音
    mySound = getMediaPath("sec1silence.wav")
    square = makeSound(mySound)

    # 设置音乐常量
    samplingRate = getSamplingRate(square) # 采样率
    seconds = 1                             # 播放一秒种

    # 设置产生方波的辅助变量
    # 每周期的秒数：确保使用浮点数

    interval = 1.0 * seconds / freq
    # 因为interval是浮点数，计算结果也将是浮点数
    samplesPerCycle = interval * samplingRate
    # 每隔半个周期需要切换一次
    samplesPerHalfCycle = int(samplesPerCycle / 2)
    sampleVal = amplitude
    s = 1
    i = 1

    for s in range(0, getLength(square)):
        # 如果已经到了半个周期
        if (i > samplesPerHalfCycle):
            # 每半个周期振幅取反一次
            sampleVal = sampleVal * -1
            # 同时重新初始化半周期计数器
            i = 0
            setSampleValueAt(square, s, sampleVal)
            i = i + 1

    return(square)
```

函数可以这样使用：

```
>>> sq440=squareWave(440,4000)
>>> play(sq440)
```

```
>>> sq880=squareWave(880,8000)
>>> sq1320=squareWave(1320,10000)
>>> writeSoundTo(sq440,getMediaPath("square440.wav"))
Note: There is no file at C:/ip-book/mediasources/
square440.wav
>>> addSounds(sq880,sq440)
>>> addSounds(sq1320,sq440)
>>> play(sq440)
>>> writeSoundTo(sq440,getMediaPath("squarecombined440.wav"))
Note: there is no file at C:/ip-book/pmediasources/
squarecombined440.wav
```

程序原理

这份菜谱创建了一段方形的波，所有样本值要么等于振幅，要么等于振幅乘以-1。现在，我们把执行`sq440 = squareWave(440, 4000)`时程序中发生的事情一步步分析一遍。

- 首先，创建了1秒钟的静音square。
- 接下来，基于采样率、频率和声音长度计算每周期的样本数目： $(1.0 * 1/440) * 22\ 050$ ，约等于50.11363。
- 计算半个周期的样本数目并转换为整数（25），这样一个周期内有一半样本值是正数，另一半是负数。用*i*来跟踪square中设置了多少样本值，以便检查是否完成了半个周期。还把sampleVal设为输入的振幅值amplitude。
- 如果完成了半个周期（`i == samplesPerHalfCycle`），就把sampleVal乘以-1得到它的相反数。如果sampleVal是正数，则它将变成负数；是负数，则会变为正数。此时也把*i*重置为0来统计接下来的半个周期。
- 把square中的样本值设为sampleVal，然后递增*i*。
- 循环结束后，返回了square声音。

你会看到，产生的波确实是方形的（如图8.4所示），而最令人吃惊的是FFT中的那些多出来的尖峰（如图8.5所示）。方波确实会带来更复杂的声音。



图8.4 440Hz的方波（左）和加法组合的方波（右）

8.5.5 三角波

下面的菜谱创建的不再是方波，而是三角波（triangular wave）。

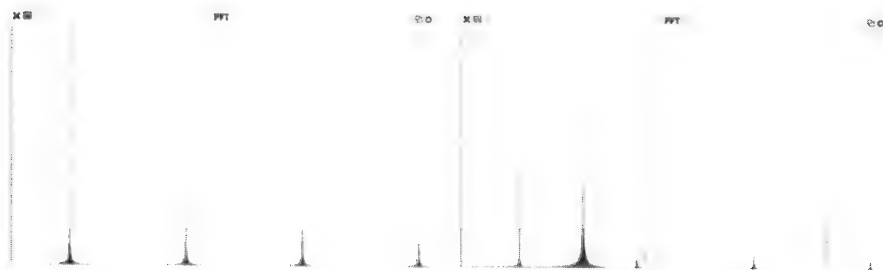


图8.5 440 Hz方波（左）和加法组合方波（右）的FFT

**程序80：产生三角波**

```
def triangleWave(freq, theAmplitude):

    # 取得一段空白声音
    mySoundF = getMediaPath("sec1silence.wav")
    triangle = makeSound(mySoundF)

    # 设置音乐常量
    # 使用参数输入的振幅
    amplitude = theAmplitude
    # 采样率（每秒钟的样本数目）
    samplingRate = 22050
    # 播放时间：1秒
    seconds = 1

    # 设置产生三角波的辅助变量
    # 每周期的秒数，确保使用浮点数
    interval = 1.0 * seconds / freq
    # 由于interval是浮点数，计算结果也将是浮点数
    samplesPerCycle = interval * samplingRate
    # 我们需要每半个周期切换一次
    samplesPerHalfCycle = int(samplesPerCycle / 2)
    # 与后续各个样本相加的值，必须是整数
    increment = int(amplitude / samplesPerHalfCycle)
    # 从波的底部开始，按需增减
    sampleVal = -amplitude
    i = 0

    # 创建1秒钟的声音
    for s in range(0, samplingRate):

        # 如果已经完成半个周期
        if(i == samplesPerHalfCycle):
            # 每隔半个周期increment取反
            increment = increment * -1
            # 同时重新初始化半周期计数器
            i = 0
```

```

sampleVal = sampleVal + increment
setSampleValueAt(triangle, s, sampleVal)
i = i + 1

play(triangle)
return triangle

```

函数可以这样使用（如图8.6所示）：

```

>>> tri440 = triangleWave(440, 4000)
>>> explore(tri440)

```



图8.6 查看三角波

程序原理

这份菜谱与产生方波的菜谱类似，但产生的波是三角形的。`sampleVal`初始化为输入振幅的相反数。每次循环中`sampleVal`都会加上一个增量（`increment`）。变量`i`跟踪我们在周期中所处的位置，每次到达周期一半时都把`increment`取负数，同时把`i`的值重置为0。

8.6 现代音乐合成

早期的音乐合成器就是基于加法合成来工作的。现在，加法合成用得不再那么多了，因为它产生的声音听起来不自然。基于录音的合成倒很常见。

如今，**频率调制合成**（Frequency Modulation Synthesis, FM合成）可能是最常见的合成技术。在FM合成中，由一个振荡器（oscillator，基于程序设定产生一系列规则输出的部件）通过其他频率来控制（调制）波的频率。产生的声音更加丰富，听上去不那么尖细，也没有明显的计算机痕迹。

另一种常见技术是**减法合成**（subtractive synthesis）。减法合成把噪声作为输入，然后使用滤镜来消除不想要的频率。结果也是音质更加丰富的输出，但通常不如FM合成那么丰富。

那为什么一定要用计算机来制作声音或音乐呢？世间有那么多悦耳的声音、美妙的音乐、伟大的音乐家，计算机制作又有什么意义呢？意义在于：假如想告诉另一个人你是如何得到这

段声音的，从而使他们能复制整个过程甚至以某种方式修改声音（可能使之更加悦耳），实现的方法就是使用程序。程序能简明地捕捉并传达一种过程——一段声音或音乐的产生过程。

8.6.1 MP3

如今，计算机上最常见的音频文件是MP3文件（或者MP4以及其他相关的或衍生的文件类型）。MP3文件是基于MPEG-3标准（实际应该是：MP3是MPEG-1/MPEG-2 Layer 3。参阅：<http://en.wikipedia.org/wiki/Mp3>。——译者注）的声音编码。它们是用特殊方法压缩的音频文件。

MP3文件使用的一种压缩方法称为无损压缩（lossless compression）。我们知道，有些技术能用更少的位存储数据。举个例子，我们知道每个样本通常占两个字节。如果不存储所有的样本，而存储样本与前一个样本的差值，那么情况会怎样呢？相邻样本之间的差值通常比-32 768~32 767这个范围小得多——它们可能在+/-1 000之间，存储起来占用的位数要少一些。

但MP3也使用有损压缩（lossy compression）。有损压缩实际会丢弃一些声音信息。举例来说，如果一段很微弱的声音紧跟着或伴随着一段很响亮的声音，那么你将听不到那段微弱的声音。模拟录音（analog recording）（唱片使用的类型）会记录所有的频率，而MP3会丢弃那些根本听不到的频率。模拟录音与数字录音的不同在于前者持续地记录声音而后者基于时间间隔来进行采样。

WAV文件也是压缩的，但不像MP3压缩得那么厉害，且只用无损技术。同样的声音，MP3文件一般比WAV格式的文件小得多。AIFF文件与WAV文件类似。

8.6.2 MIDI

乐器数字接口（Musical Instrument Digital Interface, MIDI）实际上是计算机音乐设备（如音序器、合成器、鼓乐器、电子琴等）制造商之间就设备如何协同工作达成的一套协定。使用MIDI，可以从不同的音乐键盘上控制多种合成器和鼓乐器。

MIDI更多用于编码音乐，而不是编码声音。MIDI记录的不是声音的听觉效果，而是演奏声音的方法。确切地说，MIDI编码了这样的信息：“在合成乐器X上按下音高为Y的键”，后面再跟“释放X乐器上的Y键”。MIDI的音质完全取决于合成器——产生合成乐音的设备。

MIDI文件通常很小。像“演奏音轨7上的42#键”这样的指令大约只有5字节。这使MIDI与大的声音文件相比更具吸引力。MIDI曾经在卡拉OK机上特别流行。

与MP3和WAV文件相比，MIDI有一个优点：它能以较少的字节数定义很长的音乐。但MIDI不能用于录音。举例来说，如果想录制某人演奏某种乐器的特殊风格，或者录制任何人唱歌的声音，MIDI都不适合。捕捉真实声音需要记录真实样本，因此应当使用MP3或WAV。

大多现代操作系统都内置了相当好的合成器。我们可以从Python中使用它们。JES内置了一个playNote函数，它接受一个MIDI音符、以毫秒为单位的持续时间（演奏声音的时间长度）和一个0~127之间的强度（按键的力度）值作为输入。playNote只使用钢琴风格的乐音。MIDI音符对应的是琴键而不是频率。第一音阶的C是1，C#是2。第四音阶的C是60，D是62，E是64。

以下是从JES中演奏几个MIDI音符的简单例子。我们可以用for语句在音乐中定义循环。



程序81：演奏MIDI音符（示例）

```
def song():
    playNote(60, 200, 127)
```

```

playNote(62, 500, 127)
playNote(64, 800, 127)
playNote(60, 600, 127)

```

编程摘要

<code>if</code>	Python支持决策命令。 <code>if</code> 接受一个用于测试真假的表达式（基本上，任何估值为0的表达式都为假，其他都为真）。如果测试结果为真，那么Python会执行 <code>if</code> 后面的程序块
<code>int</code>	返回输入值的整数部分，丢弃小数点之后的部分
<code>setMediaPath()</code>	可以用它选择一个文件夹来取得或保存媒体
<code>getMediaPath(baseFileName)</code>	接受一个基础文件名作为输入，返回该文件的完整路径名（假定它存在于你用 <code>setMediaPath()</code> 设置的文件夹中）
<code>playNote</code>	接受音符、持续时间和强度作为输入。每个音符用0~127之间的一个整数来表示。中央C是60。持续时间以毫秒为单位。强度也在0~127之间，如果省略，那么JES默认使用64

习题

8.1 名字解释：

1. MIDI
2. MP3
3. 模拟
4. 振幅
5. 采样率

8.2 无损压缩和有损压缩的区别是什么？哪种声音文件使用无损压缩，哪种声音文件使用有损压缩？

8.3 重写echo函数（程序69）来产生两段回声，每一段比前一段延迟delay个样本。提示：从 $(2 * \text{delay} + 1)$ 处开始下标循环，然后从 $(\text{index} - \text{delay})$ 处取得一段回声的样本，从 $(\text{index} - 2 * \text{delay})$ 处取得另一段回声的样本。

8.4 编写一个通用的混合函数，接受两段声音作为混合的输入并返回一段新的声音。它还可以接受两个数字，分别指定在混合之前从第一段声音中取用的样本数目以及用于混合的样本数目。

8.5 声音的频率加倍以后（程序72）它的长度与原来的相比有什么变化？如果每4个声音样本复制一个到目标声音，声音的长度与原来相比又有什么变化？

8.6 编写一个函数基于两段声音创建新的声音，首先取用第一段声音的一半，然后把第一段声音的后半部分跟第二段声音的前半部分相加，最后加上第二段声音的后半部分。两段声音的长度一样时做起来最容易。

8.7 编写一个函数将三段声音混合到一起。从第一段声音的一部分开始，然后是第一段声音跟第二段声音混合，然后是第二段声音跟第三段声音混合，最后是第三段声音剩下的部分。

8.8 往一段音乐中混合一些语音。开始时让音乐占75%，语音占25%，然后逐渐过渡到音乐占25%，语音占75%。

8.9 编写函数创建一段金字塔形的声波。

- 8.10 编写函数创建一段锯齿形的声波。
- 8.11 编写函数将一段声音的频率改变10次。
- 8.12 编写一个新版本的变换函数，让目标声音尽可能长。因子小于1时创建的声音应该短一些，因子大于1时创建的声音应该长一些。
- 8.13 变换函数能处理因子0.3吗？如果不能，能否把它改成将源声音中的每个样本值复制三次到目标声音中？
- 8.14 Hip-hop DJ会前后搓动唱盘，使一段声音快速来回播放。试着把“倒序播放”（程序66）跟“频率变换”（程序72）结合起来，达到与DJ搓盘同样的效果。快速播放一秒钟声音，然后再快速倒序播放一遍，这样重复两三次。（速度增加一倍不一定够，你可能需要“搓”得再快一些。）
- 8.15 考虑把频率变换程序（程序75）中的if块改成`sourceIndex = sourceIndex - getLength(source)`。这与把sourceIndex简单设置成0有什么区别？更好还是更不好？为什么？
- 8.16 如果使用因子2.0或3.0调用频率变换程序（程序75），声音会重复两遍或三遍，为什么？你能修正这个问题吗？编写一个shiftDur函数，基于参数传入的样本数目（甚至秒数）来播放声音。
- 8.17 使用声音工具找出不同乐器的特征模式。比如，钢琴发出的声音模式与人类发出的声音相反——得到的正弦波随着音调升高，振幅会降低。试着创建不同的模式，听一听它们的声音，看一看它们的波形。
- 8.18 音乐家使用加法合成时常常为声音加上包络（envelope），甚至在每一段参与叠加的正弦波上也加包络。包络的振幅随着时间变化——它可能开始时很小，然后（或快或慢地）变大，然后维持某个特定的值，最后在声音结束之前降低。这种模式有时称为起延衰（Attach-Sustain-Decay, ASD）包络。钢琴一般起得快衰得也快。笛子一般起得较慢，但可以维持任意长的时间。试着为正弦波产生器和方波产生器实现这种包络。
- 8.19 编写函数用MIDI演奏一首歌。

深入学习

好的计算机音乐书会讲到很多关于如何从零开始制作声音的内容，就像本章一样。从理解的难度上讲，Mark最喜欢的一本是Dodge和Jerse写的《Computer Music: Synthesis, Composition and Performance》[11]。计算机音乐的“圣经”是Curtis Road的大部头《The Computer Music Tutorial》[35]。

从本章讨论的层面来说，实践计算机音乐最强大的工具之一是CSound。它是一种软件音乐合成系统，免费且完全跨平台。关于如何使用CSound，Richard Boulanger的书[7]包含了你需要了解的全部内容。

构建更大的程序

本章学习目标

- 演示两种不同的设计策略：自顶向下和自底向上。
- 演示不同的测试策略：比如黑盒和白盒。
- 演示几种用于查找程序问题的调试策略。

本章计算机科学学习目标：

- 使用方法接受用户输入并为用户产生输出。
- 使用新的迭代结构：while循环。

我们之前编写的程序足以应付可用程序解决的许多问题。十几行代码的小程序可以解决许多有趣问题，带来不少有趣创新。然而，还有许多许多的问题和创新只能用更大、更复杂的程序去解决。这就是本章要讲的内容。它们是软件工程（software engineering）领域要解决的问题。

编写更大的程序需要解决一些与管理程序自身行为有关的问题。

- 你打算编写什么样的程序代码？你如何确定自己需要什么样的函数？这是一种设计过程。设计的方法有很多，最常见的两种是自顶向下设计和自底向上设计。在自顶向下设计中，首先想清楚要做什么，然后将需求细化（refine），直到自己能够确定一些代码片段，然后编写这些片段（一般从最上层开始）。在自底向上的设计中，从已知的东西开始，然后不断往上添加直到完成程序。
- 事情不会从一开始就那么顺利。有人说编程是“调试一张白纸的艺术”（the art of debugging a blank sheet of paper）[⊖]。调试就是找出运行不正常的部分，确定它为什么不正常并改正之。编程活动与调试活动密不可分。调试是一项重要技能，能帮我们弄清楚如何让程序真正运行起来。
- 即使事情从一开始就顺利，也不见得完全正确。大程序包含许多部分，需要用测试技术来保障程序中所有（或至少大部分）错误（bug）都已得到解决。
- 即使在测试和调试之后，多数大程序依然没有最后“完成”。许多大程序要经过相当长的时间不断解决时常出现的问题（比如记录和跟踪存货清单的程序）。这些大程序永远不会完工。相反，新的特性可能加入，新发现的bug必须消除。程序开发中的维护阶段会与程序使用的时间一样长。整体来讲，在一个程序的生命周期中，维护阶段明显是成本最昂贵的阶段。

9.1 自顶向下设计程序

自顶向下是大多数工程方法推荐的设计方式。首先，你需要用自然语言或数学来开发一张需求列表，即需要完成的任务，这些需求可通过迭代不断细化。细化需求就是让需求更清

[⊖] <http://foldoc.doc.ic.ac.uk/coldoc/index.html>。

晰更具体。在自顶向下设计中，细化需求的目标是达到这种效果：需求的陈述可直接实现为程序代码。

自顶向下设计广受欢迎是因为它既易于理解又便于规划——正是这样的一种设计让软件开发商业化成为可能。设想一种与客户打交道的情景：客户希望你编写某种程序，你得到了某种问题陈述（problem statement），与客户一道把它细化为一组需求。然后你开始构建程序。如果客户不满意，你可以测试一下，看软件是否满足了需求。如果它满足了需求且客户认同这种需求，那就说明你已经实现了你们的约定。如果没有满足需求，你需要让它满足——但不一定满足客户变化了的需求。

具体来讲，整个过程是这样的：

- 从问题陈述开始。如果还没有问题陈述就写一个。要说明你想做什么。
- 开始细化问题陈述。程序能否分成几个部分？是否该使用层次式分解（hierarchical decomposition）定义子函数？是否必须打开一些图片或设置一些常量值？是否有循环？需要多少循环？
- 继续细化问题陈述，直到得出语句、命令或者已经知道（或知道怎样编写）的函数。
- 编写大程序时，几乎肯定会在函数中调用其他函数（子函数）。一开始先写需要从命令区调用的函数，然后是更下层的函数，直到剩下的都是那些已经存在的子函数。

9.1.1 自顶向下设计示例

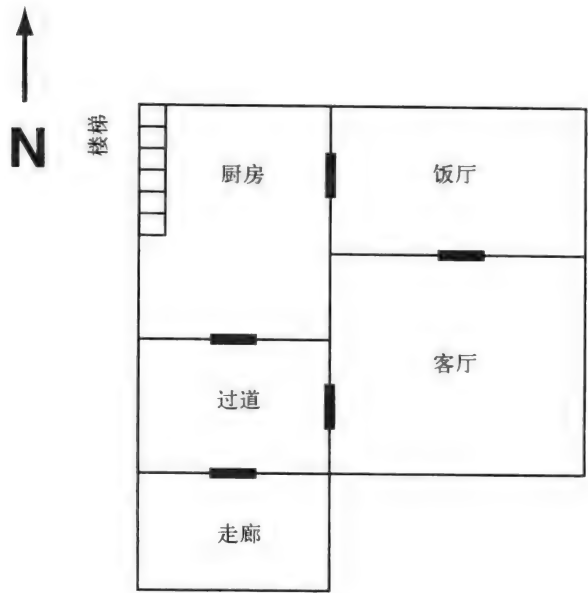
如果按照这种过程来定义函数（过程），而不是一行一行地编写代码，那我们就可以使用过程式抽象（procedural abstraction）。在过程式抽象中，我们通过调用下层函数来定义上层函数。下层的函数更易于编写、测试，上层的函数则变得更易于阅读，因为它们只是把下层函数调用一下。

在前面几章，我们已经见过一些过程式抽象的例子。其中之一就是基于复制像素的下层函数来重新定义拼贴图函数。这些下层函数是一种抽象形式。为这些代码片段起一个函数名字，我们考虑问题时就不必再基于单独的一行行代码，而是使用一个有意义的名字。再为函数提供一些参数，它就变得可重用了。

我们来构造一个简单的冒险游戏。冒险游戏是一类视频游戏，玩家在游戏中探索一个世界，使用“向北走”和“拿钥匙”之类的命令在游戏的各种空间中移动，通常需要解决一些谜题或参加一些战斗。最早的冒险游戏是1970年由William Crowther编写的，后来由Don Woods做了扩展。这款游戏基于洞穴的探索。此类风格在20世纪80年代伴随着Infocom公司的一些游戏流行起来，比如Zork和Hitchhiker's Guild to the Galaxy。现代冒险游戏多是图形化的，如Myst。早期基于文本的冒险游戏都是将玩游戏与讲故事结合起来，这也是我们的目标。

此时此刻，需要定义自己的问题了。打算构建什么样的冒险游戏呢？规模如何？玩家可以做什么？我们想在自己能够开发的限度内定义这些问题。

来构造一款让玩家在房间之间走动的冒险游戏，嗯，就是它了。没有谜题也没有真正的攻略。我们只想要一个简单例子。下面是简单的一组房间布局的草图。为了让游戏有趣一点，假定这是个令人毛骨悚然的故事场景。



9.1.2 设计顶层函数

程序如何工作呢？关于它的运行方式，我们可以想出一个概要：

- 1) 首先向玩家提供关于游戏玩法的基本知识。
- 2) 向玩家描述房间，开始时把玩家放在一个特定房间里。
- 3) 获取玩家输入的命令（“north”或“quit”）。
- 4) 根据玩家选择的命令（方向），计算它接下来要进入的房间。
- 5) 回到第2步重复执行，直到玩家说“quit”。

实际上，现在就可以编写完成这些任务的函数。不过，我们还是需要了解几个以前没见过的Python函数。

- `printNow`函数[⊖]接受一个字符串作为输入，执行时立即把参数输出到命令区。这与`print`不同，在程序结束运行之前，`print`输出的内容不会显示在命令区。`printNow`对于在游戏过程中显示内容非常有用。
- 要获得用户输入，可以使用`requestString`。`requestString`函数接受一条提示消息作为输入。它把提示消息显示在请求用户输入的窗口中（如图9.1所示）。（在Python中，从用户那里读取输入字符串的能力称为`raw_input`，原始输入。）函数返回用户输入的字符串。

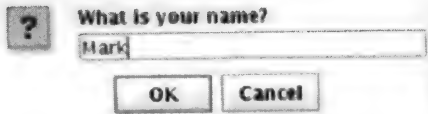


图9.1 requestString对话框的外观

[⊖] 在Python3.0中，连`print`都是函数，所以这并不奇怪。


```
>>> print requestString("What is your name?")
Mark
```

- 更大的挑战是：如何无限地持续运行一段代码直到一件特定的事情发生。我们无法用for循环实现这种需要。

相反，我们将使用一种新的循环：while循环。while循环接受一个测试（就像if）。与if不同的是，while循环无限地得重复执行循环体，直到测试条件变为假。

```
>>> x = 0
>>> while (x < 3):
...     print "Counting..."
...     x = x + 1
...
Counting...
Counting...
Counting...
```

有了这3个新函数，就可以编写与前面的概要相对应的函数了。



程序82：冒险游戏的顶层函数

```
def playGame():
    location = "Porch"
    showIntroduction()
    while not (location == "Exit") :
        showRoom(location)
        direction = requestString("Which direction?")
        location = pickRoom(direction, location)
```

这个函数与前面的概要非常接近，几乎能一行行对应起来。可能让人奇怪的是，我们还没看到或写出诸如showIntroduction、showRoom和pickRoom这样的函数。这是自顶向下设计的要点之一——远在各个部分写好之前，我们就可以规划整个程序的运行方式，并看到我们的程序中需要什么样的函数。

程序原理

- 将使用变量location来保存玩家当前所处的房间，并让玩家从“走廊”开始。
- showIntroduction函数向用户介绍这款游戏。
- 用位置（location）等于“Exit”来表示玩家离开程序的请求。只要位置不是“Exit”就继续玩游戏。
- 在每一轮循环都会显示当前房间的描述。函数showRoom显示房间的描述，使用玩家的位置（location）作为输入，以确定显示哪个房间。
- 通过requestString获得用户请求的新方向（direction）。
- 根据输入的请求方向（direction）和用户当前的房间位置（location），用pickRoom函数为用户选择新的房间。

注意，我们对这些子函数的工作细节一无所知。我们不可能马上知道它们的工作方式，因为我们还没把它们编写出来呢。也就是说，这个顶层函数，playGame，与下层子函数是解耦（decouple）的。基于输入、输出和它们应当完成的工作来定义这些子函数。现在，其他的人也可以帮我们编写这些函数。这是顶层设计在工程中使用如此广泛的另一个原因。

- 它使程序更易维护，因为可以改变不同部分而不影响整体。

- 在编写函数之前就开始规划它们。
- 基于这种规划，支持不同的程序员在同一个程序中协同工作。

9.1.3 编写子函数

既然有了规划，那么现在我们就可以编写其余的子函数，好让程序运行起来。目前我们会把所有函数放在同一个文件中，以便程序正常工作。也可以把一些有用的函数放在单独的文件中然后导入（import）这些函数。但目前还是先完成冒险游戏的设计和实现吧。

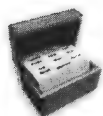


程序83：冒险游戏的showIntroduction函数

```
def showIntroduction():
    printNow("Welcome to the Adventure House!")
    printNow("In each room, you will be told which directions
    you can go.")
    printNow("You can move north, south, east, or west by typing
    that direction.")
    printNow("Type help to replay this introduction.")
    printNow("Type quit or exit to end the program.")
```

程序原理

程序介绍只是通过printNow函数向用户/玩家显示了一些信息，告诉玩家如何移动（输入一个方向），如果获得帮助以及如何退出。注意，在实现这个函数的过程中，进一步定义了后面的函数。在pickRoom函数中，必须处理诸如“help”、“quit”和“exit”这样的输入。



程序84：冒险游戏的showRoom函数

```
def showRoom(room):
    if room == "Porch":
        showPorch()
    if room == "Entryway":
        showEntryway()
    if room == "Kitchen":
        showKitchen()
    if room == "LivingRoom":
        showLR()
    if room == "DiningRoom":
        showDR()
```

程序原理

可以把showRoom做成一个长长的、含有大量printNow调用的函数。但那样的写法冗长乏味，维护起来也不容易。如果你想修改一下客厅的描述怎么办？可以在一个长长的函数中翻山越岭地找到那条需要修改的printNow。或者，可以只修改一个showLR函数。我们的程序就是这样设置的。



程序85：冒险游戏的pickRoom函数

```
def pickRoom(direction, room):
    if (direction == "quit") or (direction == "exit"):
        printNow("Goodbye!")
        return "Exit"
    if direction == "help":
        showIntroduction()
```

```

    return room
if room == "Porch":
    if direction == "north":
        return "Entryway"
if room == "Entryway":
    if direction == "north":
        return "Kitchen"
    if direction == "east":
        return "LivingRoom"
    if direction == "south":
        return "Porch"
if room == "Kitchen":
    if direction == "east":
        return "DiningRoom"
    if direction == "south":
        return "Entryway"
if room == "LivingRoom":
    if direction == "west":
        return "Entryway"
    if direction == "north":
        return "DiningRoom"
if room == "DiningRoom":
    if direction == "west":
        return "Kitchen"
    if direction == "south":
        return "LivingRoom"

```

程序原理

这个函数通过地图来定义。给定当前的房间和选定的方向，它返回玩家应该进入的新房间的名字。



程序86：在冒险游戏中显示房间

```

def showPorch():
    printNow("You are on the porch of a frightening looking house.")
    printNow("The windows are broken. It's a dark and stormy night.")
    printNow("You can go north into the house. If you dare.")

def showEntryway():
    printNow("You are in the entry way of the house. There are cobwebs in the corner.")
    printNow("You feel a sense of dread.")
    printNow("There is a passageway to the north and another to the east.")
    printNow("The porch is behind you to the south.")

def showKitchen():
    printNow("You are in the kitchen. All the surfaces are covered with pots, pans, food pieces, and pools of blood.")
    printNow("You think you hear something up the stairs that go up the west side of the room.")
    printNow("It's a scraping noise, like something being dragged along the floor.")
    printNow("You can go to the south or east.")

def showLR():
    printNow("You are in a living room. There are couches, chairs, and small tables.")
    printNow("Everything is covered in dust and spider webs.")
    printNow("You hear a crashing noise in another room.")
    printNow("You can go north or west.")

```

-这几行程序应该与下面的一行连续。Python中一条命令不可以跨多行。

```
def showDR():
    printNow("You are in the dining room.")
    printNow("There are remains of a meal on the table. You can't tell what it is,-
    and maybe don't want to.")
    printNow("Was that a thump to the west?")
    printNow("You can go south or west")
```

-这几行程序应该与下面的一行连续。Python中一条命令不可以跨多行。

[以上关于程序断行的两段“-”标注，最终应根据译本排版情况决定。不一定两页都出现]

程序原理

每个房间都有一些简单的printNow调用来描述它。从编程的角度看，这些都是非常简单的函数。从作者的角度看，这恰恰是乐趣和创新之所在。

现在，我们已经有了足以玩游戏的代码。从调用顶层函数开始，方法是在命令区输入playGame()。命令区中立即出现了游戏描述，然后一行提示出现在游戏之上的对话框中（如图9.2所示）。某次程序运行的结果如下：

```
>>> playGame()
Welcome to the Adventure House!
In each room, you will be told which directions you
    can go.
You can move north, south, east, or west by typing that
    direction.
Type help to replay this introduction.
Type quit or exit to end the program.
You are on the porch of a frightening looking house.
```



图9.2 冒险游戏的截屏

```
The windows are broken. It's a dark and stormy night.
You can go north into the house. If you dare.
You are in the entry way of the house. There are
    cobwebs in the corner.
```

```

You feel a sense of dread.
There is a passageway to the north and another to
the east.
The porch is behind you to the south.
You are in a living room. There are couches, chairs,
and small tables.
Everything is covered in dust and spider webs.
You hear a crashing noise in another room.
You can go north or west.
Goodbye!

```

9.2 自底向上设计程序

自底向上是一种不同的过程，但最终结果却是殊途同归的。开始时，对要做的事情有个大致概念——可以把它称为问题陈述。但不是去细化问题，而是关注程序方案的构建。你想尽可能地重用其他程序中的代码。

在自底向上设计中，最重要的事情是经常把自己的程序试用一下。它做了你想做的事情吗？它做了你期望的事情吗？它有意义吗？如果不是，加一些print语句，考察一下代码，直到你理解了它在做什么为止。如果你不知道它在做什么，就无法把它变成自己想要的东西。

下面是自底向上过程的一般模式，也是从一个问题陈述开始：

- 程序中有多少部分是你已经知道该如何完成的？有多少部分可以从你编写过的其他程序中获得？譬如，程序是否需要你处理声音？把本书中的几个声音菜谱用一用，你就能记起那些处理方法。程序是否要求你改变红色级别？你能找到一个完成这项功能的菜谱来使用吗？
- 现在，你能把这样一些片段（你能够编写出的或者可以从其他程序中“偷”来的）合到一起吗？如果有一些菜谱分别完成了你想要的一部分工作，那么能把它合到一起吗？
- 继续增大程序。它离你的需要更近了吗？还需要添加什么呢？
- 不断运行程序。确保它能够工作，而且你知道目前为止自己有了哪些东西。
- 重复这些过程直到你对结果满意。

自底向上过程示例

本书的多数示例都是按自底向上的过程开发的。完成背景消减和色键的方式就是很好的例子。最初我们只有“去除某人的背景并把他放进新的图片中”这样一种想法。从哪里开始呢？

我们能够想到的是，问题的一部分是找出属于人物或属于背景的全部像素。之前我们做过这样的事情，那时是找到Katie头发中的棕色以便改成红色（程序36）。这告诉我们：可以检查人物颜色与背景颜色之间是否有足够大的间距（distance）。如果有，就可以把新背景上同一点的像素颜色带进来。将图片复制到画布时，也是这样实现的。

现在，大概能编写这样的代码：

```

if distance(personColor,bgColor)<someThresholdValue:
    bgColor = getColor(getPixel(newBackground,x,y))
    setColor(getPixel(personPicture,x,y),bgColor)

```

这就是背景消减菜谱的本质。剩下的只是变量的设置（程序45）。但在那时，因为各种原因，开始尝试时我们发现效果不是很好。为此我们又转向了色键（程序46）技术。要找出哪

些像素属于前景哪些像素属于背景，色键是一种更好的方法，但基本过程与交换背景是一样的——所以多数程序都可以重用于新的上下文中。

这里的关键过程是从其他项目中获得思想（甚至代码片段）并组合它们，然后不断测试自己做的东西。自底向上编程与“调试一张白纸”非常接近。调试是自底向上设计和编程中的关键技巧。

9.3 测试程序

程序很难测试到位，对新程序员来说尤其如此：编写代码，然后就以为它做了你写的事情！只有具备高度谦逊的品质才能成就一名优秀的测试员。你必须接受这样的事实：某些东西你可能没写对，或者你还没完全理解要写什么。

测试程序有两种主要方法。一种叫白盒测试，你需要测试程序中所有可能的路径。这种方法之所以称为“白盒”，是因为你会实际查看自己的程序并考虑如何测遍程序的每一行。你了解程序的结构，于是根据这种结构来测试。

如果要对我们的冒险游戏实施白盒测试，那么可以分别从两个方向遍历每一扇门。比如，从走廊向北进入过道，然后再向南回到走廊。每次会到达正确地点吗？房间的显示正确吗？还应测试输入“help”、“quit”和“exit”命令后分别会发生什么。这可以保证我们测了程序的每一行。

另一种方法叫黑盒测试，黑盒测试中，你不需要考虑程序是怎么编写的。相反，你考虑程序的行为应该是怎样的。特别地，当分别面对有效和无效输入时，程序的响应应该是怎样的。玩家输入的命令不一定正确。玩家可能不小心拼错了命令，或者尝试了一种她觉得合理你却未曾考虑的命令。

作为例子，用黑盒方法来测一下pickRoom函数。首先，应该测试所有针对pickRoom的正确输入，就像这样：

```
>>> pickRoom('north', 'Porch')
'Entryway'
>>> pickRoom('north', 'Entryway')
'Kitchen'
```

现在，我们再试一下无效输入——拼写错误和不正确的理解。

```
>>> pickRoom('nrth', 'Porch')
>>> pickRoom('Entryway', 'Porch')
```

这的确是个问题。面对无效输入，pickRoom没返回任何东西。当试着基于它返回的值设置变量location的时候，将无法得到一个合法的房间。变量location将为空，而且玩家得不到任何有关错误的反馈。

必须修改pickRoom，让它在没有其他匹配的情况下，也必须做出合理的答复并返回适当的值。或许，最合适的返回值是同一个房间——让玩家留在原地。



程序87：冒险游戏中的pickRoom函数，改进版本

```
def pickRoom(direction, room):
    if (direction == "quit") or (direction == "exit"):
        printNow("Goodbye!")
        return "Exit"
```

```

    if direction == "help":
        showIntroduction()
        return room
    if room == "Porch":
        if direction == "north":
            return "Entryway"
    if room == "Entryway":
        if direction == "north":
            return "Kitchen"
        if direction == "east":
            return "LivingRoom"
        if direction == "south":
            return "Porch"
    if room == "Kitchen":
        if direction == "east":
            return "DiningRoom"
        if direction == "south":
            return "Entryway"
    if room == "LivingRoom":
        if direction == "west":
            return "Entryway"
        if direction == "north":
            return "DiningRoom"
    if room == "DiningRoom":
        if direction == "west":
            return "Kitchen"
        if direction == "south":
            return "LivingRoom"
    printNow("You can't (or don't want to) go in that direction.")
    return room

```

现在，程序在面对错误输入时，行为更合理一些。

```

>>> pickRoom('nrth','Porch')
You can't (or don't want to) go in that direction.
'Porch'
>>> pickRoom('Entryway','Porch')
You can't (or don't want to) go in that direction.
'Porch'

```

上面第一个句子指出输入的方向是不允许的，其中第二行（'Porch'）是玩家目前所在的房间。

测试边界条件

专业程序员会全面测试每个程序，确保程序按期望的方式工作。他们关注的黑盒项目之一是对边界条件（edge condition）的测试。程序需要处理的最小输入值是什么？要确保程序能处理最小和最大可能的输入，这就是我们说的测试边界条件的含义。

也可以将这一策略用于媒体处理程序的测试。假设图片处理程序在处理某张图片时失败了（产生了某个错误，或看上去停不下来），而且你试着跟踪程序却找不出它失败的原因。这时可以换一张不同图片试试。程序能处理小一点的图片吗？空白图片（全白或全黑）呢？或许你最终发现程序还是能够正常运行的，只是太慢了，于是在处理大图时你以为它不能工作了。

有时，处理下标的函数（比如在放大图片的程序中）在面对某种尺寸时会失败，换一种则不会。比如，镜像程序可能处理奇数个下标没有问题，却不能处理偶数个下标。碰到这种情况，

可以试试不同尺寸的输入，看看哪个成功哪个失败。

9.4 调试技巧

如果程序虽能运行却得不到想要的结果，怎样搞清楚它做了什么呢？这一过程就是调试。调试就是弄清楚程序在做什么，与你期望它做的事情有何不同，以及如何让程序从现在的样子变成你需要的样子。

最简单的调试从错误消息开始。Python提供了一些关于错误的指示，你对错误的位置（行号）也有大致概念。这些信息都能告诉你去哪里查看才能修正问题并消除错误。

更困难的调试是这样的：程序能够运行，但完成的结果不是你想要的。此时你必须搞清楚程序在做什么，而你想让它做的又是什么。

第一步永远是搞清楚程序在做什么。不管有没有错误信息，这永远是首先要做的事情。如果出现错误，重要的问题就是为什么程序只能正常工作到那里，发生错误时变量的值都是怎样的。

计算机科学思想：学会跟踪代码

调试程序时，最重要的事情就是能够跟踪代码。学会用计算机的方式思考程序。一行一行地看，弄清楚每一行在做什么。

调试时从跟踪代码开始，至少要跟踪错误前后的代码。错误信息是怎么说的？原因可能是什么？错误发生的前后，各变量的值分别是怎样的？一个有趣的问题是：为什么错误偏偏在此时发生？为什么没在程序中发生得更早一些？

如果可以，还是得运行程序。让计算机告诉你发生了什么总比你亲自弄明白要容易一点。话虽如此，可仅仅把函数运行一遍并不会得出答案。可以在代码中添加print语句来显示变量的值。

调试技巧：print语句是你的朋友

将程序中正在发生的事情打印出来。当你无法通过跟踪程序来搞清楚怎么回事的时候，可以试试这种做法。打印出复杂表达式的值。打印出像“现在执行到这个函数了！”这样的简单语句，让自己知道已经进入了你认为在调用的函数。让计算机告诉你它在做什么。

有时候，特别是在循环中，会考虑使用printNow。由于printNow在执行时立即把参数字符串打印到命令区，所以在调试中它比print更有用。你需要在一件事情发生的时候就知道它发生了。

调试技巧：不要怕修改程序

将程序保存一份副本，然后删除所有让你迷惑的部分。你能否让剩下的部分运行起来？现在开始从原始程序中把一些片段加回去（复制-粘贴）。修改程序，每次只运行一部分，这是弄清程序行为的极为有效的方法。

9.4.1 找出担心的语句

最难查找的bug是那种看上去一切正常的。代码位置错误和括号不匹配就属于此类。这种

bug在大程序中很难查找。错误信息仅仅指出某行附近的“某个地方”有问题，但Python并不确定问题到底在哪里。

要查出问题，一种省时的策略是去掉所有那些确定没有问题的语句。在你觉得OK的语句之前放一个“#”字符就可以了。如果注释掉一个if或一个for，别忘了把这条语句后面的块也注释掉。然后把程序重试一遍。

如果错误消失了，那说明你错了——你把有问题的语句给注释掉了。取消一些注释，然后再试。当错误重新回来时，它就在你刚刚取消注释的那几行语句中。

如果错误仍然存在，现在你只需检查不多的几条语句——尚未注释掉的语句。最终，要么错误消失，要么只有一条语句还没注释掉。无论是哪一种，你现在都可以找出错误的位置。

9.4.2 查看变量

除了打印以外，还有其他一些JES内置的工具能帮你搞清楚程序在做什么。`showVars`函数显示执行到该函数时所有变量和变量的值（如图9.3所示）。它显示的变量既包括当前上下文中的，也包括全局上下文中的（从命令区也可以访问到的）。可以在命令区使用`showVars()`来查看那里创建的变量——可能你已经忘记了它们的名字，或者想一次查看多个变量的值。

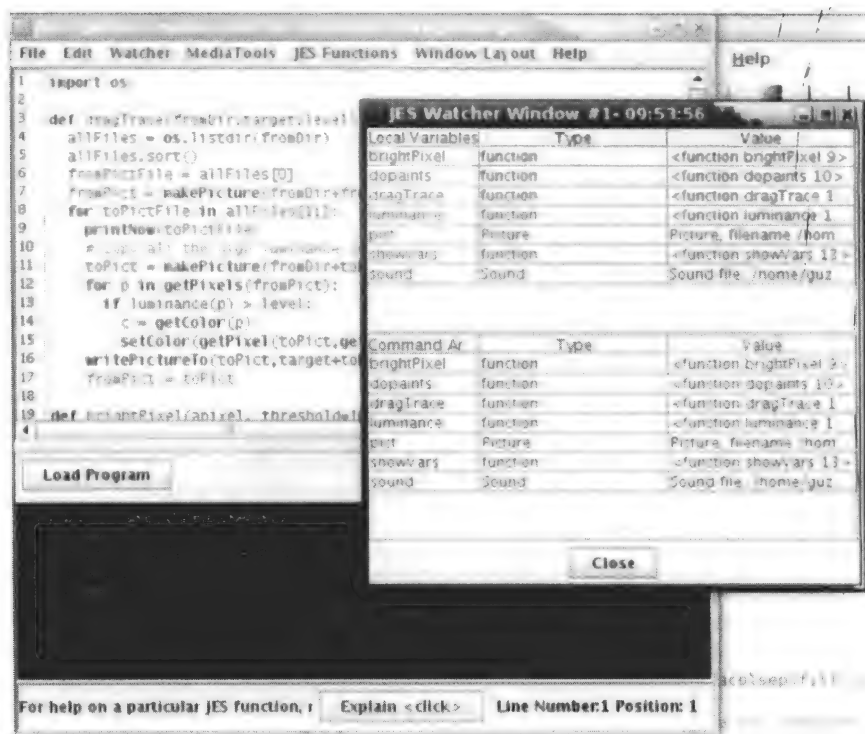


图9.3 在JES中显示变量

JES中的另一个强大工具是观察器（Watcher）。从观察器中你可以看到当前哪一行代码在执行。图9.4显示了运行下列代码时的观察器——这是程序13中的`makeSunset()`函数。只需要打开观察器（从debug菜单中或使用Watcher按钮），然后像平常一样使用命令区即可。这样每当执行自己的函数时，观察器就会出现。

```
def makeSunset(picture):
    reduceBlue(picture)
    reduceGreen(picture)

def reduceBlue(picture):
    for p in getPixels(picture):
        value=getBlue(p)
        setBlue(p,value*0.7)

def reduceGreen(picture):
    for p in getPixels(picture):
        value=getGreen(p)
        setGreen(p,value*0.7)
```



图9.4 使用观察器单步执行makeSunset()

我们可以暂停（Pause）程序的执行，然后从暂停的位置开始单步执行（Step）。我们还可以让程序停止执行（Stop），全速执行（Full Speed），甚至设置一个由慢变快的速度。观察器开着的时候，程序会运行得慢一些。程序运行得越快，显示的信息就越少（也就是说，并不是执行过的每行代码都会显示在观察器窗口中）。

除了通过单步执行程序来查看哪条语句何时执行之外，还可以观察特定的变量。点击Add Variable，JES会提示你输入一个变量名。然后，当观察器运行时，变量的值就会随着各行代码一起显示出来。即使在变量还没有值的时候你也会看到它（如图9.5所示）。

9.4.3 调试冒险游戏

让我们使用这些技术来调试一下冒险游戏。冒险游戏的问题是它基本正常！程序运行时未产生明显的错误信息。把前面说的测试流程过一遍，我们会发现程序并没有恰当地处理非法输入。但还有其他问题吗？



图9.5 使用观察器查看makeSunset()函数中的变量值

我们来看一下运行程序时的情况：

```
>>> playGame()
Welcome to the Adventure House!
In each room, you will be told which directions you
    can go.
You can move north, south, east, or west by typing
    that direction.
Type help to replay this introduction.
Type quit or exit to end the program.
You are on the porch of a frightening looking house.
The windows are broken. It's a dark and stormy night.
You can go north into the house. If you dare.
You are in the entry way of the house. There are
    cobwebs in the corner.
You feel a sense of dread.
There is a passageway to the north and another to
    the east.
The porch is behind you to the south.
You are in a living room. There are couches, chairs,
    and small tables.
Everything is covered in dust and spider webs.
You hear a crashing noise in another room.
You can go north or west.
You are in the dining room.
There are remains of a meal on the table. You can't
    tell what it is, and maybe don't want to.
Was that a thump to the west?
You can go south or west
You are in the kitchen. All the surfaces are covered
    with pots, pans, food pieces, and pools of blood.
You think you hear something up the stairs that go up
    the west side of the room.
It's a scraping noise, like something being dragged
    along the floor.
You can go to the south or east.
Goodbye!
```

· 有什么问题吗？以下是程序输入中的两个问题：

1) 用户很难通过回头看程序输出来搞清楚哪个房间在哪儿。房间的描述非常模糊。

2) 我们也很难回头找到自己在哪里输入过什么。如果这是一幅有很多房间的大地图，我们更想回滚到前面看看自己输入过什么，以便进入不同的房间。

首先解决第一个问题。我们需要在房间描述之间使用某种空白或分隔字符。这样的一条语句该放在哪里呢？当然，它应该位于顶层的主循环之内。可以放在showRoom函数中，比如写到第一行；或者，也可以在顶层函数中，紧挨着放在显示房间语句的前面或后面。出于细节调整最好置于子函数中的考虑，我们还是修改showRoom吧。



程序88：冒险游戏中的showRoom函数，改进版本

```
def showRoom(room):
    printNow("=====")
    if room == "Porch":
        showPorch()
    if room == "Entryway":
        showEntryway()
    if room == "Kitchen":
        showKitchen()
    if room == "LivingRoom":
        showLR()
    if room == "DiningRoom":
        showDR()
```

现在解决第二个问题。应该把玩家输入过的命令打印出来，而且必须在调用requestString之后做这件事。这次的修改还是可以放在顶层循环中，另外也可以放在pickRoom开头。两种都能达到目的。但这一次我们将做出与上次相反的选择，pickRoom函数已经相当复杂了。我们将在玩家做出选择之后立即回应。



程序89：冒险游戏中的playGame函数，改进版本

```
def playGame():
    location = "Porch"
    showIntroduction()
    while not (location == "Exit") :
        showRoom(location)
        direction = requestString("Which direction?")
        printNow("You typed: "+direction)
        location = pickRoom(direction, location)
```

现在我们可以试试问题解决之后的程序。

```
>>> playGame()
Welcome to the Adventure House!
In each room, you will be told which directions you
    can go.
You can move north, south, east, or west by typing
    that direction.
Type help to replay this introduction.
Type quit or exit to end the program.
=====
You are on the porch of a frightening looking house.
The windows are broken. It's a dark and stormy night.
```

```

You can go north into the house. If you dare.
You typed: north
=====
You are in the entry way of the house. There are
cobwebs in the corner.
You feel a sense of dread.
There is a passageway to the north and another to
the east.
The porch is behind you to the south.
You typed: east
=====
You are in a living room. There are couches, chairs,
and small tables.
Everything is covered in dust and spider webs.
You hear a crashing noise in another room.
You can go north or west.
You typed: north
=====
You are in the dining room.
There are remains of a meal on the table. You can't
tell what it is, and maybe don't want to.
Was that a thump to the west?
You can go south or west
You typed: west
=====
You are in the kitchen. All the surfaces are covered
with pots, pans, food pieces, and pools of blood.
You think you hear something up the stairs that go up
the west side of the room.
It's a scraping noise, like something being dragged
along the floor.
You can go to the south or east.
You typed: south
=====
You are in the entry way of the house. There are
cobwebs in the corner.
You feel a sense of dread.
There is a passageway to the north and another to
the east.
The porch is behind you to the south.
You typed: exit
Goodbye!

```

9.5 算法和设计

算法是过程的一般性描述，可通过任何特定的编程语言实现。算法相关的知识是专业程序员工具箱中的工具之一。到目前，已经见过了多种算法：

- 采样算法（sampling algorithm）是这样一种过程，它可以把声音的频率提高或降低，也可以把图片缩小或放大。描述采样算法不需要讨论循环或者源坐标和目标坐标的递增。采样算法的原理是调整从源向目标复制样本或像素的方式——除了简单取用各个样本或像素，还可以每两个样本/像素取一个、每个样本/像素取两次，或者使用其他采样模式。
- 我们见过如何从源向目标复制像素或样本，只需要分别跟踪在源和目标中的位置。

- 我们还见过像素和样本的混合（融合）本质上是一样的。对参与混合的每个像素或样本应用一个权值，然后把加权的結果相加，从而创建混合的声音或融合的图片。

算法在程序设计中的角色是允许我们在基础的程序代码之上抽象出一种程序设计的描述。专业程序员知道很多算法，这使他们可以从更高的层次思考程序设计问题。可以直接讨论图片的颜色反转和反转后图片的镜像，不必讨论循环、源下标和目标下标。可以关注“镜像”之类的抽象名称而不必关注代码。

程序员还知道很多与算法有关的知识。他们知道如何让算法更高效，什么时候它们不适合，以及什么地方可能出毛病。比如，我们知道缩放声音时必须小心，防止越过声音的边界。从执行的快慢和对内存的需求方面考虑，算法有优劣之分。关于算法的速度，将在第13章有更多讨论。

9.6 在JES之外运行程序

Python程序可通过多种方式运行。如果构建更大更复杂的程序，可能考虑在JES之外运行它们。本书教授的内容可以直接在Python（或Jython）中使用。for和print这样的命令在Python和Jython中使用都没问题。ftplib和urllib这样的系统库在Python和Jython中也是完全一样的。

然而，使用的媒体工具却不是Python或Jython内置的。但它们可以在Python中使用。有些Python实现，比如Myro (<http://www.roboteducation.org>)，包含与JES一样的图像函数。另外，还有一个Python图形处理库（Python Imaging Library, PIL），它提供了类似功能，但使用不同的函数名字。

可以使用我们为Jython提供的库。Jython可以从<http://www.jython.org>获取，在多数计算机系统中都可以使用。只需要几条额外的命令，我们的媒体库就可以在Jython中使用（如图9.6所示）。

以下是让媒体函数在传统Jython中可用的方法。

- 找出相关的导入（import）模块，Python使用变量sys.path（存在于内置的系统库sys中）来罗列它查找模块的目录。如果想在Jython中使用JES媒体库，需要把那些模块文件所在的位置放到sys.path变量中。（这就是JES中的setLibraryPath完成的工作。）

你需要通过import sys来获得访问sys.path变量的能力。为了处理这个变量，用insert方法把JES Sources目录加到系统路径中（如图9.6所示）。在Macintosh上，需要在JES应用程序中引用Java和Jython代码，方法是使用类似这样的命令：
sys.path.insert(0, "/users/guzdial/JES.app/Contents/Resources/Java")。

- 然后就可以使用from media import *, 从而可以在Jython中使用pickAFile和makePicture之类的函数了。

实际上，在你每次按下Load按钮的时候，from media import *语句都会被插入到程序区中（对你（学生程序员）来说，是不可见的）。JES特有的媒体特性就是这样提供给你的程序的。

以下是在Linux中产生如图9.6所示图像的方法：

```
guzdial@guzdial-laptop:~$ jython
Jython 2.2.1 on java1.6.0_07
Type "copyright", "credits" or "license" for more
information.
```

```
>>> import sys
>>> sys.path.insert(0, "/media/MyUSB/jes-4-0/Sources")
>>> from media import *
>>> p = makePicture(pickAFile())
>>> show(p)
```

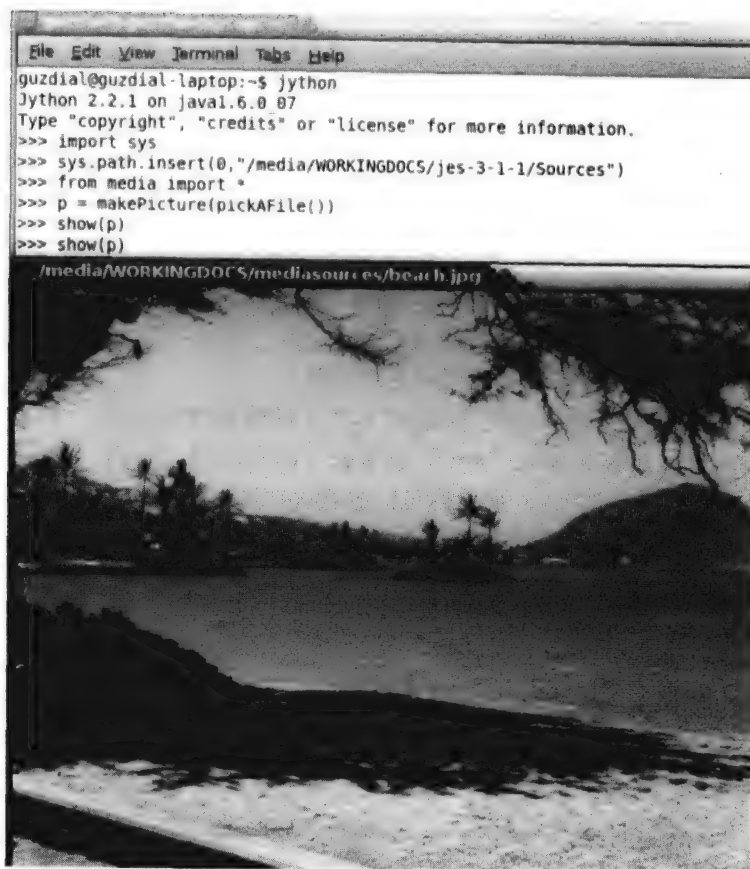


图9.6 使用Jython在JES之外调用媒体函数

编程摘要

<code>println</code>	在程序运行时，将输入参数立即打印出来。与 <code>print</code> 不同， <code>println</code> 打印的内容要到程序停止运行之后才会显示
<code>requestString</code>	显示一个带有输入提示的对话框，接受用户输入的字符串并返回这个字符串。JES中还有其他类似的函数，如 <code>requestNumber</code> 、 <code>requestInteger</code> ，甚至 <code>requestIntegerInRange</code> （将输入的整数限制在两个参数值之间）
<code>showVars</code>	显示程序中已存在的所有变量以及它们的值
<code>while</code>	只要语句中指定的测试条件为真，就一直循环执行 <code>while</code> 之后的语句块

习题

9.1 通常，人们会在一个程序成功运行且经过全面的调试、测试之后才优化（optimize）它

(使之运行得更快或占用更少的内存)。——当然,在每次优化改动之后,必须再测试一次。以下是针对冒险游戏的一种优化。showRoom将room变量与每一个可能的房间进行比较——即使之前的比较已经匹配。Python提供了一种只匹配一次的方法:elif,只需把后面的if都改成elif。elif语句的意思是“else if”。只有在前面的if为假的情况下,才会测试elif语句。一句if之后可以有任意多条elif语句。比如可以这样使用:

```
if (room == 'Porch'):
    showPorch()
elif (room == 'Kitchen'):
    showKitchen()
```

使用elif重写showRoom函数以优化之。

- 9.2 像pickRoom这样使用一大堆嵌套if语句的函数阅读起来很麻烦。用适当的注释解释各部分代码完成的工作(检查房间,检查某房间里可能的移动方向)可以使它清晰一些。请为pickRoom加上注释,使之更易于阅读。
- 9.3 为所有的函数加上注释,使他人阅读起来更加方便。
- 9.4 为玩家添加另一个变量hand并将它初始化为空。修改程序的描述,在客厅中添加一把“钥匙”(key)。玩家在客厅输入“key”命令就可以把钥匙(key)拿到手中。现在,如果玩家进入厨房时持有钥匙key,那么就可以走楼梯,向“西”走并沿楼梯上楼。为实现这种效果,需要为游戏再添加一些房间。
- 9.5 设计从走廊“下去”(down),探索地下秘密世界的功能。
- 9.6 添加其他秘密道具,玩家可以拿到这些道具以便进入不同房间,比如进入走廊下面的通道时可以使用的灯笼。
- 9.7 在饭厅中添加一颗炸弹。玩家在饭厅输入“bomb”就可以把炸弹拿到手中。一只怪物住在楼上,玩家在楼上输入“bomb”就可以把炸弹扔过去,怪物顿时灰飞烟灭。
- 9.8 添加玩家输掉游戏的功能(可能玩家死掉了)。输掉时游戏应该打印出发生了什么,然后退出。比如,发现怪物时如果玩家手里没有炸弹,那他就会输掉游戏。
- 9.9 添加玩家打赢游戏的功能。打赢时游戏应该打印出发生了什么,然后退出。比如,在走廊下找到秘密宝藏会让玩家赢得游戏。
- 9.10 房间不一定全部用文字描述,还可以在玩家进入某个房间时播放一段相关的声音。最好使用play函数来播放,这样玩家可以在声音播放的同时继续游戏。
- 9.11 除了使用文字和声音以外,房间的描述还可以视觉化。进入一个房间时,可以显示一幅相关的图片。还可以更进一步,每幅图片只显示一次,再次进入此房间时把图片重画(repaint)一下,带回到前景中即可。
- 9.12 这款冒险游戏有一处潜在的问题,房间的名字在多个地方重复出现。如果饭厅在一个地方拼成“DiningRoom”,在另一个地方却拼成了“DinngRoom”(少了第二个i),那么游戏运行时就可能不正常。添加的房间越多,输入房间名字的地方就越多,出现错误的可能性也就越大。

有多种方法可以减小发生这种错误的可能性:

- 改用数字表示房间的名字,而不用字符串。输入并检查“4”总比输入并检查“DiningRoom”更简单一些。
- 使用一个变量DinningRoom表示饭厅,然后其他地方都基于这个变量来检查位置。这样

一来，使用数字还是字符串就无所谓了（但字符串阅读和理解起来都容易得多）。

从上面的方法中选用一种，重写冒险游戏，从而减少潜在的错误。

- 9.13 `printNow`函数不是程序运行过程中向用户呈现信息的唯一方法。还可以使用`showInformation`函数，此函数接受一个字符串作为输入并将它显示在对话框中。目前`showRoom`调用的子函数们假定我们会使用`printNow`来显示房间信息。如果像`showPorch`这样的函数能返回描述房间的字符串，那么`showRoom`函数既可以用`printNow`也可以用`showInformation`来显示房间描述。

试着改写显示房间描述的函数，让它们返回字符串，然后修改`showRoom`函数，让它能在两种方式之间方便地变换——既可以在命令区打印房间信息，也可以用对话框显示房间信息。

- 9.14 考虑下面的程序：

```
def testMe(p,q,r):
    if q > 50:
        print r
    value = 10
    for i in range(1,p):
        print "Hello"
        value = value - 1
    print value
    print r
```

如果执行`testMe(5, 51, "Hello back to you!")`，会输出什么结果？

- 9.15 `def newFunction(a, b, c):`
 `print a`
 `list1 = range(1,5)`
 `value = 0`
 `for x in list1:`
 `print b`
 `value = value +1`
 `print c`
 `print value`

如果输入

```
newFunction("I", "You", "walrus")
```

来调用上面的函数，计算机上会输出什么结果？

深入学习

如今的文本式冒险游戏常被称为交互式小说（interactive fiction）。有一些网站和网络资料库可以下载交互式小说，你可以下载一些来玩玩。更开心的是，有一些编程语言是专门用于设计和构建视频游戏的，比如Inform 7。

Frederick P. Brooks写的《The Mythical Man-Month: Essays on Software Engineering (2nd Edition)》(Addison-Wesley, 1995)可能是迄今为止软件工程领域最好的一本书。Brooks指出，软件开发中的许多问题都是组织问题。

文本、文件、网络、数据库和单媒体

第10章 创建和修改文本

第11章 高级文本技术：Web和信息

第12章 产生Web文本

创建和修改文本

本章学习目标

本章媒体学习目标：

- 创建信函样式的文本。
- 处理结构化文本，如电话地址簿。
- 产生随机的结构化文本。

本章计算机科学学习目标：

- 使用点号语法访问对象的成员部件。
- 处理字符串。
- 读写文件。
- 理解树状文件结构。
- 编写处理程序的程序，这引出了一种强大思想：使用解释器和编译器。
- 使用Python标准库中的模块，如random和os工具集。
- 了解Python标准库有哪些可用功能。
- 拓宽对import功能的理解。

10.1 文本作为单媒体

麻省理工学院（MIT）媒体实验室的创立者Nicholas Negroponte说过，计算机实际是单媒体（unimedia），而正是这一事实让基于计算机的多媒体成为可能。事实上，计算机只理解一样东西：0和1。可以把计算机用于多媒体，是因为任何媒体都可以编码成0和1。

他也可以说文本作为单媒体的观念。可以把任何媒体编码成文本。与0和1相比，使用文本有一处更好的地方：文本可以阅读。在本章稍后的内容中，将把声音映射成文本，然后再把文本映射回声音，对图片也会做同样操作。一旦有了用文本表示的媒体，我们就不必再回到原先的媒体：可以将声音映射成文本，然后再映射成图片，这样就可以创建视觉化的声音了。

万维网中首要的东西就是文本。随便访问一张网页，然后到Web浏览器菜单中选择“查看源文件”，此时你看到的就是文本。实际上每个页面都是文本。文本引用了你查看页面时得到的图片、声音和动画，但页面本身是用文本定义的。文本中的词语用一种称为超文本标记语言（HyperText Markup Language, HTML）的语法来定义。

到目前为止，我们一直使用有限的几种编程语言概念来编写程序。使用JES，我们仅通过赋值、for、if、print、return和函数就能完成各种声音和图片程序。但编程语言所具备的特性和功能要比这多得多。本章将展现位于JES底层的東西，从而教会你更多编程技能。

10.2 字符串：构造和处理字符串

通常，本文基于字符串来处理。字符串是字符组成的序列。字符串以数组的形式存储在内存里，就像声音一样。字符串仿佛一个个信箱连续存储在内存中——各个信箱紧挨着放在一起。比如，字符串“Hello”会存放在紧挨着的五个信箱中：一个信箱保存“H”对应的二进制码，下一个保存“e”，再下一个保存“l”，依此类推。

定义字符串时，使用的方法是在字符序列的两端分别加上引号。Python有一处不同寻常的地方：它允许使用多种引号形式定义字符串。可以使用单引号、双引号，甚至三引号。引号可以嵌套。定义字符串时如果使用双引号开头，那么字符串内部就可以出现单引号，因为字符串一直到下一个双引号才结束。如果用单引号开头，则可以在字符串中随意使用双引号，因为Python会等待一个单引号来结束字符串。

```
>>> print 'This is a single-quoted string'
This is a single-quoted string
>>> print "This is a double-quoted string"
This is a double-quoted string
>>> print """This is a triple-quoted string"""
This is a triple-quoted string
```

为什么会有三引号？因为它允许我们在字符串中嵌入换行、空格、跳格等字符。我们很难在命令区方便地使用三引号，但在程序区完全可以。

```
def sillyString():
    print """This is using triple quotes. Why?
    Notice the different lines.
```

```
Because we can do this."""
```

```
>>> sillyString()
This is using triple quotes. Why?
Notice the different lines.
```

```
Because we can do this.
```

使用这么多引号形式的价值在于我们可以方便地在字符串中放置引号。比如，双引号在HTML中有特定的语法含义。如果要写一个创建HTML页面的Python函数（Python的常见用途之一），就需要含有引号的字符串。由于这三种引号都可以定义字符串，所以只要用单引号来标记字符串的起止位置，就可以在其中嵌入双引号。

```
>>> print " "
Your code contains at least one syntax error, meaning
it is not legal jython.
>>> print ' "'
"
```

字符串可以看成字符序列的数组。它的确是个序列——可以用for来遍历所有字符。

```
>>> for i in "Hello":
...     print i
...
H
e
l
l
o
```

在内存中，字符串是一列连续的信箱（继续使用我们把内存比做邮政室的隐喻）。每个信箱包含相应字符的二进制码。函数ord()可以给出各个字符的美国信息交换标准码（American Standard Code for Information Interchange, ASCII）。于是，我们会看到字符串“Hello”有5个信箱，第一个包含72，然后是101、108……

```
>>> str = "Hello"
>>> for char in str:
...     print ord(char)
...
72
101
108
108
111
```

如果是在JES中，上面的叙述就略显简单了。我们使用的Python版本，Jython，是基于Java构建的，而Java并不使用ASCII编码字符串。Java使用Unicode，这是一种每个字符使用两个字节的字符编码方式。两个字符提供了65 536种可能的组合。多出来的这些编码远远超出了简单的拉丁字母、数字和标点符号的需要。我们还可以表示平假名、片假名，以及其他象形文字系统。

说这些是为了告诉你各种字符的数目比键盘能直接输入的要多得多。除了特殊符号之外，还有一些不可见的字符，比如跳格，以及按回车键产生的字符。在Python（以及许多其他语言，如Java和C）字符串中，使用反斜杠转义（backslash escape）来输入这些字符。反斜杠转义就是反斜杠（\）后面再跟一个字符。

- \t等同于按跳格键。
- \b等同于按回退键（放在字符串中没有多大用处，但可以使用）。打印\b时，大多数系统中会显示一个小方块——它实际上是不可打印的字符（见后面的图片）。
- \n等同于按回车键。
- \uXXXX，其中XXXX是字符0~9和A~F组成的编码（称为十六进制数），表示该数码所代表的Unicode字符。你可以从网址：<http://www.unicode.org/charts>查阅这些编码。

下面是交互运行一些Python命令的截图，从中可以看到一些Unicode字形。

```
>>> print "hello\tthere,\nMark"
hello   there
Mark
>>> print u"\uFEED"
,
>>> print u"\u03F0"
π
>>> print "This\u2013is\u2014a\u2014test"
This_ is
a_ test
```

还记得之前我们在文件名字符串开头使用过的“r”吗？比如：

```
r"C:\ip-book\mediasources\barbara.jpg"
```

“r”要求Python以原始模式（raw mode）读取字符串。所有的反斜杠转义都被忽略。这在Windows路径名中很重要，因为Windows使用反斜杠做路径分隔符。举例来说，如果文件名以字母“b”开头，Python将把\b视为回退字符，而不是分隔符再跟一个b。

可以用“+”运算符把字符串加在一起（也称为字符串的连接），使用函数len()获得字符串的长度。

```
>>> hello = "Hello"
>>> print len(hello)
5
>>> mark = ", Mark"
>>> print len(mark)
6
>>> print hello+mark
Hello, Mark
>>> print len(hello+mark)
11
```

10.3 处理部分字符串

使用方括号（[]）语法来引用字符串的一部分。

- string[n]返回字符串中第n个字符，注意首字符是第0个。
- string[n:m]返回字符串从第n个字符开始，一直到但不包括第m个字符的片段（与range()函数类似）。你可以选择性地省略n或m。如果省略n，函数会假定从0开始。如果省略m，则假定一直到字符串结束。也可以在任何一端使用负数，从而从那一端剪除相应数量的字符。

我们可以把字符串中的字符想象成盒子，每个盒子都有自己的下标。

H	e	l	l	o
0	1	2	3	4

```
>>> hello = "Hello"
>>> print hello[1]
e
>>> print hello[0]
H
>>> print hello[2:4]
ll
>>> print hello
Hello
>>> print hello[:3]
Hel
>>> print hello[3:]
lo
>>> print hello[:]
Hello
```

```
>>> print hello[-1:]
o
>>> print hello[:-1]
Hell
```

10.3.1 字符串方法：对象和点号语法简介

实际上，Python中的一切东西都不只是一个值——它们都是对象（object）。对象将数据（比如一个数字、一个字符串或一个列表）与操作对象的方法（method）结合在一起。方法类似于函数，但无法从全局域访问它们。不能像执行pickAFile()或makeSound()那样执行一个方法，方法是只能通过对象来访问的函数。

Python中的字符串是对象。它们不只是字符序列——还拥有方法。这些方法不能从全局域访问，只有字符串知道它们。使用点号（.）语法来执行字符串中的方法：即输入object.method()。

例如，capitalize()就是字符串专有的方法之一。这个方法把调用方法的字符串转换成首字母大写。它不能在函数或数字上使用。

```
>>> test="this is a test."
>>> print test.capitalize()
This is a test.
>>> print capitalize(test)
A local or global name could not be found. You need
to define the function or variable before you try to
use it in any way.
NameError: capitalize
>>> print 'this is another test'.capitalize()
This is another test
>>> print 12.capitalize()
Your code contains at least one syntax error, meaning
it is not legal jython.
```

实用的字符串方法有很多：

- 如果字符串以给定的前缀（prefix）开始，那么startswith(prefix)返回真。别忘了Python中1或更大的数值是真（true），0是假（false）。

```
>>> letter = "Mr. Mark Guzdial requests the pleasure of your company..."
>>> print letter.startswith("Mr.")
1
>>> print letter.startswith("Mrs.")
0
```

- 如果字符串以给定后缀（suffix）结尾，那么endswith(suffix)返回真。endswith在检查文件名是否为程序所需的类型时特别有用。

```
>>> filename="barbara.jpg"
>>> if filename.endswith(".jpg"):
...     print "It's a picture"
...
It's a picture
```

find(str)、find(str, start)，以及find(str, start, end)都是在目标字符串中查找

`str`并返回`str`的起始下标。在后两种可选形式中，你可以告诉方法从哪个下标开始查找，甚至到何处停止查找。

`find()`方法失败时会返回-1，这一点非常重要。为什么是-1？因为0或更大的数都有可能找到待查字符串时的合法下标。

```
>>> print letter
Mr. Mark Guzdial requests the pleasure of your company...
>>> print letter.find("Mark")
4
>>> print letter.find("Guzdial")
9
>>> print len("Guzdial")
7
>>> print letter[4:9+7]
Mark Guzdial
>>> print letter.find("fred")
-1
```

另外，还有一个`rfind(findstring)`函数（以及同样的带有可选参数的版本）从字符串的尾部向前查找。

- `upper()`将字符串变成大写。
 - `lower()`将字符串变成小写。
 - `swapcase()`将大写字母变小写，小写字母变大写。
 - `title()`将各单词首字符变成大写，其余的变成小写。
- 这些方法可以串联起来——一个方法修改的结果再交由另一个方法修改。

```
>>> string="This is a test of Something."
>>> print string.swapcase()
tHIS IS A TEST OF sOMETHING.
>>> print string.title().swapcase()
tHIS iS a tEST oF sOMETHING.
```

- `isalpha()`在字符串非空且只含有字母时返回真——不能含有数字和标点。
- `isdigit()`在字符串非空且只含有数字时返回真。你可以用这个函数检查搜索的结果。假如你正在编写一个查询股价的程序，想解析出当前价格，而不是股票名称。如果解析出现问题，那么程序可能做出你不希望的交易。这种情况下可以用`isdigit()`来自动检查结果。
- `replace(search, replace)`在字符串中搜索`search`字符串并将它们替换为`replace`字符串。它将结果返回，而不会修改原来的字符串。

```
>>> print letter
Mr. Mark Guzdial requests the pleasure of your company...
>>> letter.replace("a","!")
'Mr. M!rk Guzd!al requests the ple!sure of your comp!ny...'
>>> print letter
Mr. Mark Guzdial requests the pleasure of your company...
```

10.3.2 列表：强大的结构化文本

列表（list）是一种强大的结构，可以把它想象成一种结构化文本。列表用方括号定义，

各元素之间以逗号分隔，但它可以包含任何东西——包括子列表。与字符串一样，可以用方括号语法引用它的一部分，也可以用“+”运算符把不同列表连接起来。列表也是序列，因此可以用for循环遍历其中的元素。

```
>>> myList = ["This","is","a", 12]
>>> print myList
['This', 'is', 'a', 12]
>>> print myList[0]
This
>>> for i in myList:
...     print i
...
This
is
a
12
>>> print myList + ["Really!"]
['This', 'is', 'a', 12, 'Really!']
>>> anotherList=["this","has",["a",["sub","list"]]]
>>> print anotherList
['this', 'has', ['a', ['sub', 'list']]]
>>> print anotherList[0]
this
>>> print anotherList[2]
['a', ['sub', 'list']]
>>> print anotherList[2][1]
['sub', 'list']
>>> print anotherList[2][1][0]
sub
>>> print anotherList[2][1][0][2]
b
```

列表有一些方法是字符串所不具有的。

- `append(something)`将一样东西 (`something`) 放入列表末尾。
- `remove(something)`从列表中删除一样东西 (`something`) ——如果它存在的话。
- `sort()`将列表元素按字母表顺序排列。
- `reverse()`将列表反序。
- `count(something)`告诉你列表中有多少个`something`。
- `max()`和`min()`就是我们之前见过的接受一个列表作为输入并分别返回其中最大值、最小值的函数。

```
>>> list = ["bear","apple","cat","elephant","dog","apple"]
>>> list.sort()
>>> print list
['apple', 'apple', 'bear', 'cat', 'dog', 'elephant']
>>> list.reverse()
>>> print list
['elephant', 'dog', 'cat', 'bear', 'apple', 'apple']
>>> print list.count('apple')
2
```

`split(delimiter)`是最重要的字符串函数之一，它根据你提供的分隔符 (`delimiter`) 字符串把一个字符串转换成一个子串列表。有了它，字符串就可以转换成列表。

```
>>> print letter.split(" ")
['Mr.', 'Mark', 'Guzdial', 'requests', 'the',
'pleasure', 'of', 'your', 'company...']
```

可以用`split()`来处理格式化文本——不同部分之间以明确定义的字符串隔开的文本，比如，电子工作表中以跳格分隔的文本或以逗号分隔的文本。下面是一个用结构化文本保存电话簿的例子。电话簿各行之间以换行符分隔，各列之间以冒号分隔。可以先按换行符切分再按冒号切分，从而获得一个由子列表组成的列表。搜索这样一个列表用简单的`for`循环就可以做到。



程序90：一个简单的电话簿应用

```
def phonebook():
    return """
Mary:893-0234:Realtor:
Fred:897-2033:Boulder crusher:
Barney:234-2342:Professional bowler:"""

def phones():
    phones = phonebook()
    phonelist = phones.split('\n')
    newphonelist = []
    for list in phonelist:
        newphonelist = newphonelist + [list.split(":")]
    return newphonelist

def findPhone(person):
    for people in phones():
        if people[0] == person:
            print "Phone number for",person,"is",people[1]
```

程序原理

这里一共有三个函数：一个提供电话文本，另一个创建电话列表，第三个查询电话号码。

- 第一个函数，`phonebook`，创建结构化文本并返回它，程序中使用了三引号，这样文本中可以用换行符来格式化不同的行。格式为：姓名、冒号、电话号码、冒号、职务，然后是冒号和行结束符。

```
>>> print phonebook()

Mary:893-0234:Realtor:
Fred:897-2033:Boulder crusher:
Barney:234-2342:Professional bowler:
```

- 第二个函数，`phones`，返回所有电话组成的列表。它访问`phonebook`产生的电话表，把它切分成不同的行。`split`的结果是含有冒号的字符串组成的列表。然后，循环中再用`split`基于冒号对各个列表做进一步切分。最后，`phones`函数返回的是一个列表的列表。

```
>>> print phones()
[[''], ['Mary', '893-0234', 'Realtor', '']],
[['Fred', '897-2033', 'Boulder crusher', '']],
[['Barney', '234-2342', 'Professional bowler', '']]
```

- 最后，第三个函数，`findPhone`接受一个名字作为输入并找到相应的电话号码。它循环遍历`phones`函数返回的所有子列表，找到其中第一个（下标为0的）与输入名字相同的

那个子列表，最后输出结果。

```
>>> findPhone('Fred')
Phone number for Fred is 897-2033
```

10.3.3 字符串没有字体

字符串并没有**字体**（字母的外观特征）或**样式**（通常是粗体、斜体、带下划线和其他一些可用于字符串的效果）与之关联。字体与样式信息都是通过字处理器或其他程序附加到字符串上的。通常这些信息被编码成**样式串**（style run）。

样式串是字体与样式信息的独立表示，通常会使用指向字符串内部的下标数字来指明生效位置。例如，“The old *brown* fox runs”的样式可以编码成：[[bold 0 6][italics 8 12]]。

考虑一下带有样式串的字符串。你如何称呼这种相关信息的组合？它显然不是单个的值。可以将样式串与字符串一起编码成一个复杂的列表吗？当然可以——使用列表，我们什么都能做到。

大多数管理格式化文本的软件都会把把样式串与字符串一起编码成一个**对象**。对象关联着可以分成多个部分的数据（如字符串和样式串）。对象知道如何使用方法来操作自己的数据，这些方法只有对象自己知道。如果多个对象都知道某个方法名字，那么它们可能完成同样的事情，但完成的方式不一样。

这些只是铺垫。关于对象的内容以后会讨论。

10.4 文件：存放字符串和其他数据的地方

文件是磁盘上的大量字节组成的命名集合。文件通常有一个**基本名**和一个**文件后缀**。（注意，这与前面章节中使用的“基本名”一词的含义不完全一致。——译者注）文件BARBARA.JPG具有基本名“barbara”和文件后缀“jpg”，后者告诉我们这个文件是JPEG图片。

文件以目录（有时也称为文件夹）的形式集中起来。目录除了包含文件之外也可以包含其他目录。计算机上有一个称为**根目录**的基础目录。在一台使用Windows操作系统的计算机上，基础目录就是像“C:\”这样的目录。关于从基础目录开始要访问哪些目录才能到达一个特定文件的完整描述称为一条**路径**。

```
>>> file=pickAFile()
>>> print file
C:\ip-book\mediasources\640x480.jpg
```

打印出来的路径告诉我们如何从根目录开始找到文件640X480.JPG。从“C:\”开始，选择目录ip-book，然后选择目录mediasources。

我们把这种结构称为**树**（如图10.1所示），把“C:\”称为树的**根**。树的分支上是子目录。任何一个目录都可以包含更多目录（分支）或文件，文件称为**叶子**。除了根以外，树上的任何一个**结点**（分支或叶子）都有单一的父分支结点，但一个父结点可以有多个**孩子**分支或叶子。

为处理文件，需要了解文件和目录，特别是大量的文件。面对大的网站时，需要处理的文件数目会很大。处理视频时，针对每秒视频都要大约处理30个文件（每帧画面）。你肯定不愿意针对每一帧都编写一行代码来打开它。你会考虑编写程序来遍历目录结构，从而处理网页和视频文件。

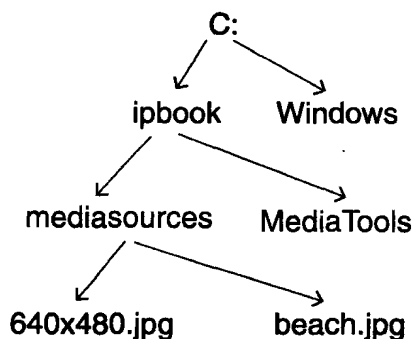


图10.1 目录树示意图

还可以把树表示成列表。列表可以包含子列表，这正如目录可以包含子目录，因此用列表编码目录非常容易。关键的一点是：可以用列表来表示像树这样复杂的、层次式的关系（如图10.2所示）。

```

>>> tree = [["Leaf1", "Leaf2"], ["Leaf3"], ["Leaf4"],
"Leaf5"]]
>>> print tree
[['Leaf1', 'Leaf2'], [['Leaf3'], ['Leaf4'], 'Leaf5']]
>>> print tree[0]
['Leaf1', 'Leaf2']
>>> print tree[1]
[['Leaf3'], ['Leaf4'], 'Leaf5']
>>> print tree[1][0]
['Leaf3']
>>> print tree[1][1]
['Leaf4']
>>> print tree[1][2]
Leaf5
  
```

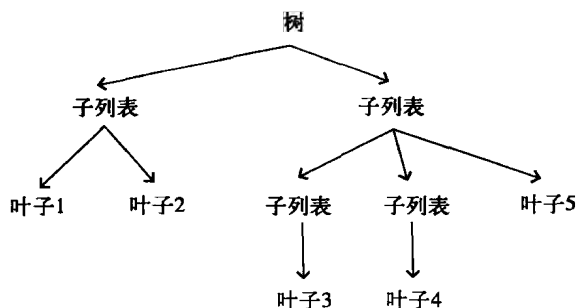


图10.2 列表示意图

10.4.1 打开文件和操作文件

打开文件是为了读写。打开文件的函数就叫`open(filename, how)`（不奇怪吧）。文件名（`filename`）可以是完整路径，也可以只有基础名加后缀。如果不提供路径，文件将从JES的当前目录打开。

`how`是一个字符串，描述你想用文件做什么。

- “rt”表示“以文本方式读文件——帮我把字节翻译成字符”。
- “wt”表示“以文本方式写文件”。
- “rb”和“wb”分别表示“读字节”和“写字节”。处理二进制文件（如JPEG、WAV、Word和Excel文件）时你会用到这两种。

函数open()返回一个文件对象，之后你就可以用它操作文件。文件对象知道如下的一组方法：

- file.read()将整个文件读成一个巨大的字符串。（不要试图读取以写方式打开的文件。）
- file.readlines()将整个文件读入一个列表，列表中的每个元素对应文件的一行。每次文件打开之后，只能使用一次read()或readlines()。
- file.write(something)将something写入文件。（不要试图写入以读方式打开的文件。）
- file.close()关闭文件。如果执行了写操作，那么关闭它可以确保所有内容都写到磁盘上去。如果执行了读操作，那么它会释放操作文件所用的内存。不论哪种情况，文件用完之后都应该关闭一下。一旦关闭了文件，就不能再对它执行读写了，除非再次打开它。下面的例子打开了我们之前编写的一个程序并将它作为字符串读入，之后又读入列表中。

```
>>> program=pickAFile()
>>> print program
C:\ip-book\programs\littlePicture.py
>>> file=open(program,"rt")
>>> contents=file.read()
>>> print contents
def littlePicture():
    canvas=makePicture(getMediaPath("640x480.jpg"))
    addText(canvas,10,50,"This is not a picture")
    addLine(canvas,10,20,300,50)
    addRectFilled(canvas,0,200,300,500,yellow)
    addRect(canvas,10,210,290,490)
    return canvas
>>> file.close()
>>> file=open(program,"rt")
>>> lines=file.readlines()
>>> print lines
['def littlePicture():\n', '
canvas=makePicture(getMediaPath("640x480.jpg"))\n', '
addText(canvas,10,50,"This is not a picture")\n', '
addLine(canvas,10,20,300,50)\n', '
addRectFilled(canvas,0,200,300,500,yellow)\n', '
addRect(canvas,10,210,290,490)\n', ' return canvas']
>>> file.close()
```

下面的例子写了个无聊的文件。\\n用于在文件中换行。

```
>>> writeFile = open("myfile.txt","wt")
>>> writeFile.write("Here is some text.")
>>> writeFile.write("Here is some more.\\n")
>>> writeFile.write("And now we're done.\\n\\nTHE END.")
>>> writeFile.close()
>>> writeFile=open("myfile.txt","rt")
>>> print writeFile.read()
Here is some text.Here is some more.
And now we're done.
```

```
THE END.
>>> writeFile.close()
```

10.4.2 制作套用信函

我们不仅可以编写程序拆分结构化文本，还可以编写程序组装结构化文本。大家都很熟悉的一种经典的结构化文本就是垃圾邮件，或者说套用信函。真正优秀的垃圾邮件作者（如果听上去不矛盾的话）会在消息中加入一些真正提到你的细节。他们是如何做到的呢？非常简单——他们有一个接受相关输入并插入到适当位置的函数。



程序91：套用信函产生器

```
def formLetter(gender, lastName, city, eyeColor):
    file = open("formLetter.txt", "wt")
    file.write("Dear ")
    if gender=="F":
        file.write("Ms. "+lastName+":\n")
    if gender=="M":
        file.write("Mr. "+lastName+":\n")
    file.write("I am writing to remind you of the offer ")
    file.write("that we sent to you last week. Everyone in ")
    file.write(city+" knows what an exceptional offer this is!")
    file.write("(Especially those with lovely eyes of"+eyeColor+"!)")
    file.write("We hope to hear from you soon.\n")
    file.write("Sincerely,\n")
    file.write("I.M. Acrook, Attorney at Law")
    file.close()
```

程序原理

这个函数接受性别、姓氏、城市和眼睛的颜色作为输入。它打开formLetter.txt文件，写下一个根据接收人性别调整的开头，然后写出一大串文本，将输入参数中的信息插入到适当的位置上。最后函数关闭了文件。

使用formLetter("M", "Guzdial", "Decatur", "brown")命令执行此函数时，它产生出了：

```
Dear Mr. Guzdial:
I am writing to remind you of the offer that we
sent to you last week. Everyone in Decatur knows what
an exceptional offer this is!(Especially those with
lovely eyes of brown!)We hope to hear from you soon.
Sincerely,
I.M. Acrook,
Attorney at Law
```

10.4.3 编写程序

现在我们开始使用文件。我们的第一个程序将做一件非常有趣的事情——一个程序修改另一个程序。读取littlePicture.py文件（程序50）并修改插入到图片中的文本字符串。我们会找到（find()）addText()函数调用，然后在其中查找两个双引号。最后我们写一个新文件，其内容首先是littlePicture.py文件到第一个双引号之前的部分，然后插入新的字符串，最后是littlePicture.py文件剩下的部分：从第二个引号一直到结尾。



程序92：用程序修改littlePicture程序

```
def changeLittle(filename, newString):
    # 获得原始文件的内容
    programFile = "littlePicture.py"
    file = open(programFile, "rt")
    contents = file.read()
    file.close()
    # 现在找到安放新字符串的恰当位置
    addPos = contents.find("addText")
    # addText之后的双引号
    firstQuote = contents.find("'", addPos)
    # firstQuote之后的双引号
    endQuote = contents.find("'", firstQuote + 1)
    # 创建新文件
    newFile = open(filename, "wt")
    newFile.write(contents[:firstQuote + 1]) # 包括引号
    newFile.write(newString)
    newFile.write(contents[endQuote:])
    newFile.close()
```

程序原理

该程序打开了littlePicture.py文件（因为没有提供路径，所以这样的名字假定文件位于JES使用的目录中）。程序将整个文件读成一个大大的字符串，然后关闭了文件。之后程序使用find方法找到了addText的位置，待替换字符串开头和结尾的双引号就在addText函数调用中。然后程序打开一个新文件（“wt”表示“writable text”，可写文本），先写littlePicture程序中找到的第一个双引号之前的内容，然后写参数字符串，再写littlePicture程序下一个双引号之后的部分。这样，程序便替换了插入图片中的文本。最后，程序关闭了新文件。

若使用changeLittle("sample.py", "Here is a sample of changing a program")命令来运行这一函数，将得到如下的sample.py文件：

```
def littlePicture():
    canvas=makePicture(getMediaPath("640x480.jpg"))
    addText(canvas,10,50,"Here is a sample of changing a program")
    addLine(canvas,10,20,300,50)
    addRectFilled(canvas,0,200,300,500,yellow)
    addRect(canvas,10,210,290,490)
    return canvas
```

基于向量的画图程序就是这样工作的。在AutoCAD、Flash或Illustrator中修改一条直线的时候，实际修改的是底层的图片表示——实际上就是一段小程序，它运行的时候能产生你所处理的图片。修改直线的时候，实际修改的就是这段程序，然后这段程序重新执行便产生了更新后的图片。这一过程会不会太慢？计算机快得很，我们根本注意不到。

处理文本的能力对于从因特网上收集数据来说非常重要。因特网上的内容大多数是文本。打开你常见的网页，然后点击菜单中的“查看源文件”（View Source）（或其他类似的）选项。你在浏览器中看到的网页就是用这样的文本来定义的。后面我们将学习如何直接从因特网上下载文件，但现在我们暂时假定你已经从因特网上保存（下载）了一些页面到文件或磁盘中，这

样我们可以只搜索磁盘上的内容。

比如，在因特网上你可以找到与某些生物（比如寄生虫）有关的核苷酸序列。我找到过这种类型的文件，内容就像下面一样：

```
>Schisto unique AA825099
gcttagatgtcagattgagcacgatgatcgattgaccgtgagatcgacga
gatgcgagatcgagatctgcatacagatgatgaccatagtgtacg
>Schisto unique mancons0736
ttctcgctcacactagaagcaagacaattttacactattattattattatt
accattattattattattattactattattattattattactattatttta
ctacgtcgctttttcactccctttatttctcaaattgtgtatccttccttt
```

假定有一组序列（比如“ttgtgta”），想知道它是哪种寄生虫的一部分。如果将上面的文件读到一个字符串中，我们就可以在其中搜索子序列。如果能找到（即find结果不等于-1），那么我们就从找到的位置反向搜索位于每个寄生虫名字之前的“>”字符，然后再向后找到行结尾（换行符），这样便取得了寄生虫的名字。若找不到这个子序列（find返回-1），那么就说明它不在这个文件中。



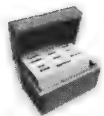
程序93：在寄生虫核苷酸序列中查找子序列

```
def findSequence(seq):
    sequencesFile = getMediaPath("parasites.txt")
    file = open(sequencesFile, "rt")
    sequences = file.read()
    file.close()
    # 查找序列
    seqLoc = sequences.find(seq)
    # print "Found at:", seqLoc
    if seqLoc <> -1:
        # 现在，查找序列所对应名字之前的">"
        nameLoc = sequences.rfind(">", 0, seqLoc)
        # print "Name at:", nameLoc
        newline = sequences.find("\n", nameLoc)
        print "Found in ", sequences[nameLoc:newline]
    if seqLoc == -1:
        print "Not found"
```

程序原理

findSequence函数接受一段序列作为输入。它打开parasites.txt文件（在setMediaPath指定的媒体文件夹下）并将整个文件读入字符串sequences中。使用find在字符串sequences中查找那段序列。如果找到（即find的结果不为-1），便从找到序列的位置（seqLoc）反向一直到字符串开头（0）的范围内查找“>”，因为每一组序列都是以“>”开始的。然后再从大于号开始向后找到行结尾（“\n”）。这样便给出了在原来的sequences中能找到输入子序列那个寄生虫名字的位置。

现在，我们试过了将全部文件内容读入一个大字符串，然后再处理这个字符串。然而，如果文件非常大，那么一次处理一行会更好一些。可以使用readlines函数达到目的，readlines返回一个字符串的列表，其中每个字符串对应文件的一行。举例来说，如果想把文件中出现的某个字符串全部改掉，可以使用下面的函数。



程序94：将文件中出现的某个单词全部替换

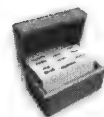
```
def replaceWord(fileName,origWord,repWord):
    file = open(fileName,"rt")
    outFile = open("out-" + fileName,"wt")
    for line in file.readlines():
        newLine = line.replace(origWord,repWord);
        outFile.write(newLine)
    file.close()
    outFile.close()
```

有些程序在因特网上四处游荡，不断从Web页面中收集信息。比如，Google新闻页面 (<http://news.google.com>) 上的内容并不是Google的记者写的，Google有一些程序四面出击，不断抓取其他新闻站点的标题。这些程序是如何工作的呢？它们不过是从因特网上下载页面，然后从中提取自己想要的片段。

举个例子，假如说你想编写一个函数来读取当地的气象网页，从而提供当前气温。在亚特兰大，<http://www.ajc.com/weather>（《Atlanta Journal-Constitution》网站的气象页面）就是不错的一个网页。看一下源文件，我们可以找到页面上显示气温的地方，以及为提取气温所需了解的上下文关键特征。在某一天Mark找到的气象页面中，相关部分如下：

```
<td ><font size=-2><br>
</font>
<font size="-1"
face="Arial, Helvetica, sans-serif"><b>Currently
</b><br>
Partly sunny<br> <font size="+2">54<b>&deg;</b><br>
</font>
<font face="Arial, Helvetica, sans-serif" size="+1">
F</font></font></td> </tr>
```

可以看到，里面有个单词Currently，而气温就在°这几个字符之前。假定气象网页保存在名为ajc-weather.html的文件中，我们可以编写一个程序来提取这些片段并返回气温。但这个程序不可能永远适用于最新的AJC气象页面。页面格式可能变化，我们查到的关键文本可能移位或消失。但只要页面格式不变，我们的菜谱就可以正常工作。



程序95：从气象页面上获取气温信息

```
def findTemperature():
    weatherFile = getMediaPath("ajc-weather.html")
    file = open(weatherFile,"rt")
    weather = file.read()
    file.close()
    # 查找气温
    currLoc = weather.find("Currently")
    if currLoc <= -1:
        # 现在找出气温后面的"<b>&deg;"
        tempLoc = weather.find("<b>&deg;",currLoc)
        tempstart = weather.rfind(">",0,tempLoc)
        print "Current temperature:",weather[tempstart+1:tempLoc]
    if currLoc == -1:
        print "They must have changed the page format--can't find the temp"
```

程序原理

这个函数假定文件ajc-weather.html存放在setMediaPath指定的媒体文件夹下。findTemperature函数打开文件并以文本方式读取内容，然后关闭文件。先查找单词“Currently”。如果找到（find的结果不为-1），再在“Currently”的位置（保存在变量currLoc中）之后查找度数符号。然后再反向寻找前一个标签的结尾，即“>”字符。气温就在这两点之间。如果currLoc为-1，就放弃搜索，因为找不到单词“Currently”了。

10.5 Python标准库

每种编程语言都会有一种用新功能扩展语言基础功能的方法。在Python中，这一功能称为**模块导入**。之前我们见过，**模块（module）**就是一个定义了新功能的Python文件。导入（import）一个文件就如同在import语句的位置输入了那个文件，于是文件中所有的对象、方法和变量一下子都有了定义。

Python带有一个功能全面的模块库，你可以用它做各种各样的事情，如访问因特网、产生随机数、访问目录中的文件——开发Web页面或处理视频时，访问目录文件的能力非常有用。

让我们把访问目录中的文件作为第一个例子。我们需要使用os模块。在os模块中，列举目录中各个文件的函数叫listdir()。我们使用点号（.）语法访问模块中的东西。

```
>>> import os
>>> print os.listdir("C:\ip-book\mediasources\pics")
['students1.jpg', 'students2.jpg', 'students5.jpg',
'students6.jpg', 'students7.jpg', 'students8.jpg']
```

可以用os.listdir()为目录中的图片文件加上名字标题，或在图片中插入一句文本，比如版权声明。listdir()只返回基础文件名和后缀，这足以确保我们处理的是图片而不是声音或其他东西，但它不能给出makePicture()所需要的完整路径。要得到完整路径，可以将输入目录与listdir()返回的基础文件名结合起来——但需要在两者之间加上路径分隔符。Python有一条规范：如果文件名中含有“/”，那么它会被替换成符合当前操作系统的路径分隔符。



程序96：为目录中的一组图片加上标题

```
import os

def titleDirectory(dir):
    for file in os.listdir(dir):
        print "Processing:", dir+"/"+file
        if file.endswith(".jpg"):
            picture = makePicture(dir+"/"+file)
            addText(picture, 10, 10, "Property of CS1315 at Georgia Tech")
            writePictureTo(picture, dir+"/"+file+"titled-"+file)
```

程序原理

titleDirectory函数接受一个目录（以字符串表示的路径名）作为输入。它遍历目录中的每个文件名file。如果文件名以“.jpg”结尾，那它很可能是一幅图片。于是我们便基于给定目录dir中的文件file构造一个图片对象。我们在图片中添加了文本，最后把图片写回dir目录中，并在文件名之前加上“titled-”作为新的文件名。

10.5.1 再谈导入和私有模块

实际上，import语句有多种形式。我们在这里使用的import module会导入模块中的所有实体，使它们都可以用点号语法来引用。此外还有其他几种用法：

- 可以从一个模块中导入几样东西，然后不必用点号语法就可以访问它们。这种形式是：

```
from module import name.
```

```
>>> from os import listdir
>>> print listdir(r"C:\Documents and Settings")
['Default User', 'All Users', 'NetworkService',
 'LocalService', 'Administrator', 'Driver',
 'Mark Guzdial']
```

我们可以用from module import *来导入模块中的所有实体，而且不必使用点号语法就可以访问它们。

- 如果我们想导入一个模块并使用新的名字来引用它，可以用import module as newname，其中newname就是新的名字。使用这种方法可以更方便地从Jython中访问Java库。Java库中有一些很长的名字，比如java.awt.event，我们可以用这种语法创建名字的简写，比如：

```
import java.awt.event as event
```

然后，就可以用event引用java.awt.event中的元素了。

模块就是一个Python文件。之前我们也见到过，可以把自己编写的代码作为模块导入。如果你有一个函数findTemperature位于JES目录下的findTemperatureFile.py文件中，那么只需执行import findTemperature from findTemperatureFile就可以使用findTemperature函数了，就好像文件被录入到程序区中一样。

10.5.2 另一个有趣模块：Random

另外一个有趣的、有时也很有用的模块是random。其中的函数random.random()产生0到1之间（平均分布）的随机数。

```
>>> import random
>>> for i in range(1,10):
...     print random.random()
...
0.8211369314193928
0.6354266779703246
0.9460060163520159
0.904615696559684
0.33500464463254187
0.08124982126940594
0.0711481376807015
0.7255217307346048
0.2920541211845866
```

随机数可用于从列表中随机选取单词这样的任务，而且会很有趣。random.choice()函数就能完成这种任务。

```
>>> for i in range(1,5):
...     print random.choice(["Here", "is", "a",
"list", "of", "words", "in", "random", "order"])
```

```
...
list
a
Here
list
```

在此基础上，我们可以产生随机的句子，方法是从列表中随机地选取名词、动词和短语。



程序97：随机产生话语

```
import random

def sentence():
    nouns = ["Mark", "Adam", "Angela", "Larry", "Jose", "Matt", "Jim"]
    verbs = ["runs", "skips", "sings", "leaps", "jumps", "climbs", "argues", "giggles"]
    phrases = ["in a tree", "over a log", "very loudly", "around-
the bush", "while reading the newspaper"]
    phrases = phrases + ["very badly", "while skipping", "instead-
of grading", "while typing on the CoWeb."]
    print random.choice(nouns), random.choice(verbs),
    random.choice(phrases)
```

这几行程序应该与下面的一行连续。Python中一条命令不可以跨多行。

程序原理

我们只是创建了一些名词、动词和短语的列表——应注意所有的组合在单、复数方面要符合英语语法。print语句定义了目标结构、一个随机的名词、一个随机的动词，再加一个随机的短语。

```
>>> sentence()
Jose leaps while reading the newspaper
>>> sentence()
Jim skips while typing on the CoWeb.
>>> sentence()
Matt sings very loudly
>>> sentence()
Adam sings in a tree
>>> sentence()
Adam sings around the bush
>>> sentence()
Angela runs while typing on the CoWeb.
>>> sentence()
Angela sings around the bush
>>> sentence()
Jose runs very badly
```

这里使用的基本过程在仿真（simulation）程序中很常见。我们在程序中定义了一种结构：定义了哪些词可以看成名词、动词或短语，也声明了结果的样子——一个名词接一个动词再接一个短语。然后，我们使用随机选择来填充这种结构。这一程序会引出一些有趣的问题：基于这种结构和随机数，一共可以模拟多少个句子？可以用这种方法做智能仿真吗？智能仿真与真正能思考的计算机之间又有什么区别呢？

设想有一个程序读取来自用户的输入并产生一个随机的句子。程序中可以有的一些规则，用

来搜索输入中的关键字并给出相应的回答，就像：

```
if input.find("Mother") <> -1:
    print "Tell me more about your Mother"
```

Joseph Weizenbaum在很多年前就写过这么一个程序，程序的名字叫*Doctor*（后来被称为*Eliza*）。他的程序可以冒充一个罗杰式的精神治疗专家（Rogerian psychotherapist），针对你说的话做出响应，响应带有随机性，但会从你的话中搜索关键词，显得自己像是真的在听一样。这个程序当初只是个玩笑，并不是真的要创造智能仿真技术。令Weizenbaum惊诧的是：人们开始严肃地看待这个程序，把它看成真正的治疗专家。Weizenbaum后来却改变了自己的研究方向，不再研究人工智能，转而关注使用技术的伦理，以及人们究竟有多么容易被技术欺骗。

10.5.3 Python标准库的例子

到目前为止，我们已经见过了像os、sys和random这样的Python标准模块。Python标准库（Python Standard Library）中有大量的模块。基于很多理由，你应该使用这些模块：

- 这些模块写得很好——速度快、文档翔实、测试充分。你完全可以信任它们并节省自己的精力。
- 重用程序代码永远是个好主意。你应当养成重用代码的良好习惯。
- 在自底向上的设计过程中，从现有模块开始是启动一个新项目的不二法门。

下面列出了一些有必要考察一下的模块：

- datetime和calendar模块知道如何操控日期、时间和日历。比如，可以算出1976年签署《美国独立宣言》的那天是星期几。

```
>>> from datetime import *
>>> independence = date(1976, 7, 4)
>>> independence.weekday()
3
>>> # 0表示星期一，因此3是星期四
```

- math模块知道许多重要的数学函数，如sin（正弦函数）和sqrt（平方根函数）。
- zipfile模块知道如何读写压缩的“zip”文件。
- email模块提供了编写程序处理电子邮件（如编写垃圾邮件过滤器）所需要的设施。
- SimpleHTTPServer实际上是个独立的Web服务器——并且是Python可编程的！

编程摘要

通用程序片段

random	产生随机数或进行随机选择的模块
os	与操作系统打交道的模块

字符串函数和程序片段

string[n], string[n:m]	返回字符串中位于位置n([n])的字符或n~(m - 1)的子串。记住在这些函数中，下标从0开始
------------------------	--

(续)

startswith	如果字符串以输入的字符串开头，则返回真
endswith	如果字符串以输入的字符串结尾，则返回真
find	如果字符串中可以找到输入字符串则返回其下标，否则返回-1
upper, lower	返回分别转换成大写或小写的新字符串
isalpha, isdigit	分别在整个字符串全是字母或数字（十进制）字符的情况下，返回真
replace	接受两个输入子串——将给定字符串中出现的第一个子串的所有实例全部替换成第二个子串
split	使用输入的字符串做分隔符，将字符串分解成子串列表

列表函数和程序片段

append	将输入值附加到列表尾部
remove	从列表中删除输入值
sort	列表排序
reverse	列表反序
count	统计输入值在列表中出现的次数
max, min	给定一个数字列表作为输入，分别返回列表中的最大值和最小值

习题

- 10.1 创建一个包含句子 “Don't do that!” 的字符串变量。创建一个包含双引号的字符串变量。创建一个包含跳格符的字符串变量。创建一个含有文件名的字符串变量，文件名中要有反斜杠。
- 10.2 编写一个函数，将某个字符串中的字符每隔一个打印一个。
- 10.3 编写一个函数，接受一个字符串，以相反的顺序打印其中的字符。
- 10.4 编写一个函数，在给定字符串中找出输入字符串的第二个实例并删除之。
- 10.5 编写一个函数，将输入句子中的各个单词每隔一个处理一个，处理的动作是转换成大写。如果输入的句子是 “The dog ran a long way”，函数将输出 “The DOG ran A long WAY”。
- 10.6 编写一个函数，接受一个句子和一个单词索引，将索引对应的单词转换成大写后再返回这个句子。举例来说，如果函数接受了句子 “I love the color red” 和下标4，则返回 “I love the color RED”。
- 10.7 编写一个函数，接受一个句子作为输入，将单词顺序扰乱后返回。举例来说，如果输入的句子是 “Does anything rhyme with orange?” 函数可能返回 “Orange with does anything rhyme?”。
- 10.8 编写一个函数，从一个按字段分隔表示的地址字符串中找出某人的邮政编码。比如，函数可能读入一个字符串，其中包含 “name:line1:line2:city:state:zipCode” 这样的格式，然后函数返回其中的邮政编码（zipCode部分）。
- 10.9 将changeLittle函数改为使用readlines，而不是read。
- 10.10 编写一个函数，为某个目录中的所有图片制作更小的版本（缩略图）。应该让用户输入目录路径和缩放因子。

- 10.11 使用ord将一个字符串中的所有字符编码成一个数字，以此来创建一条秘密消息。
- 10.12 编写一个函数将列表中的各个项目反序。
- 10.13 编写一个函数，接受一个字符串列表，一个输入字符串和一个新字符串，将列表中出现的所有输入字符串全部替换成新字符串。
- 10.14 random.random()函数中返回的数字是真正随机的吗？它们是如何产生的？
- 10.15 将Joseph Weizenbaum的Eliza程序找来看看。看自己能否写出一个类似的程序，但只限提问跟学校有关的问题。
- 10.16 简单回答以下问题。
- (a) 有些任务你会通过编写程序来完成，有些任务你不会考虑编写程序来完成，分别给出一个例子。
 - (b) 数组、矩阵和树的区别是什么？每一种结构都曾用于表示某种数据，分别给出一个例子。
 - (c) 点号语法是什么，什么时候使用它？
 - (d) 为什么红色不适合用于色键技术？
 - (e) 函数和方法的区别是什么？
 - (f) 为什么磁盘上的文件更适合用树表示而不是用数组？为什么磁盘上会有这么多目录，而不是一个巨大的目录？
 - (g) 基于向量的图像表示与位图图像表示（如JPEG、BMP和GIF）相比有哪些优势？
- 10.17 在程序20中我们见过镜像图片的代码，在程序67中我们见过镜像声音的代码。用同样的算法镜像文本想来也不会太难。编写一个函数，接受一个字符串并返回镜像的字符串：把前半部分复制到后半部分。
- 10.18 扩展套用信函菜谱，增加一种宠物的名字和类型，然后在信函中引用宠物。基于"Your pet " + petType + ", " + petName + " will love our offer!"这样的结构可以产生"Your pet poodle, Fifi, will love our offer!"。
- 10.19 设想你有班上所有同学的性别组成的列表（每个元素用一个字符表示）。列表的样子应该像“MFFMMMFFMFMFFFM”这样，其中M表示男性、F表示女性。编写一个函数percentageGenders(string)，接受一个表示性别的字符串，然后分别统计其中M和F的数目并打印两者各占的比例。举例来说，如果输入的字符串为“MFFF”，函数应打印出这样的结果：“There are 0.25 males, 0.75 females.”。（提示：某些数值最好乘以1.0来确保获得浮点数而不是整数）。
- 10.20 你为了作业忙到深夜，却没注意自己的手指错误地按在了另一排键上，这样写了学期论文中好长的一段。
- 你本来是想输入：“This is an unruly mob.”，实际却输入了：“Ty8s 8s ah 7hr7o6 j9b.”。基本上，U成了7，I成了8，O成了9，P成了0，J成了U，K成了I，L成了O，N成了H，而M成了J。（敲错的键只有这些——在错误走得更远之前你及时发现了。）还好你从来没碰过shift键，所以只需要关心小写字母。
- 作为一名掌握了Python语言的人，你决定快速编写一个程序来修正这段文本。编写函数fixItUp，接受一个字符串作为输入，返回各字符回归本位的新字符串。
- *10.21 写一个函数doGraphics，接受一个列表作为输入。doGraphics函数首先基于mediasources文件夹下的640×480.JPG文件创建一张画布。你将根据输入列表中的命

令在画布上作图。列表中的各个元素是命令字符串，这些字符串分为两类：

- “b 200 120”表示在 $x=200$ ， $y=120$ 的位置，即(120, 200)画一个黑点。数字当然会变化，但命令永远是“b”。你可以假定输入的坐标都是三位数。
- “1 000 010 100 200”表示从(0, 10)到(100, 200)画一条直线。

比如，某个输入列表可能是这样的：`["b 100 200", "b 101 200", "b 102 200", "1 102 200 102 300"]`。(元素数目可以是任意的)。

深入学习

Mark用来趟过Python模块之河的一本书是Frederik Lundh的《Python Standard Library》(Python标准库)[29]。另外，可以从如下网址找到库模块的列表及其文档：<http://docs.python.org/library/>。

高级文本技术：Web和信息

本章学习目标

本章媒体学习目标：

- 编写程序直接访问因特网上的文本信息。
- 将声音或图片翻译成文本，然后再将文本译回声音或图片。
- 在图片中隐藏消息。

本章计算机科学学习目标：

- 使用程序访问因特网。
- 演示信息可通过多种方式来编码。

11.1 网络：从Web获取文本

当计算机相互通信时便形成了网络。计算机内部使用电压来编码0和1，但网络通信很少通过线路上的电压来实现。长距离维持特定电压十分困难。相反，在网络通信中，0和1使用其他方式来编码。比如，调制解调器（modulator-demodulator, modem）将0和1映射成不同的音频频率。如果听到这些不同的音调，我们会觉得像嗡嗡的蜜蜂，而对调制解调器来说，它是纯二进制的。

就像洋葱和怪物，（相关典故可能来自电影《Shrek》（怪物史瑞克）中的台词：“Ogres are like onions, they have layers.”（怪物就像洋葱，它们有层次。——译者注）网络也是分层的。最底层是物理基质：信号是如何传递的？更高的层次定义了数据编码的方式：0是如何构造的？1呢？一次只发送一位吗？还是一次发送一组字节组成的分组（packet）？更高层次上还会定义通信协议。一台计算机如何告诉另一台计算机它想与它通信？通信的内容是关于什么的？如何寻址一台计算机？基于这些独立而清晰的层次考虑问题，并保持它们的独立与清晰，我们就可以方便地替换其中一部分而不改变其他部分。比如，大多数使用直接网络连接的人都用一根网线连接到以太网（Ethernet），而以太网实际上是一种中层协议，在无线网络中同样可以使用。

人类自身也使用协议。如果Mark向你走来，伸出手对你说：“嗨，我叫Mark。”很可能你也会伸出手来说：“我叫Carolina。”（假定你的名字是Carolina——如果不是就太好玩了。）每一种文化中都会有关于如何相互问候的协议。计算机协议与此类似，只不过它们落在纸上，可以精确地传达相关过程。协议中说的话也很类似，一台计算机可能发送消息“HELO”给另一台计算机来启动一次会话（我们不清楚协议的作者为何不能不省略那个L从而把单词拼对），也可以发送“BYE”来终止会话。（我们有时甚至直接把计算机协议的启动过程称为“握手”。）它的全部工作就是建立一条连接，并确保双方都理解正在发生的事情。

因特网（Internet）是网络的网络。如果你家里有一台设备（比如路由器）能让多台计算机彼此通信，那你就有了一个网络，很可能你已经可以在计算机间复制文件或共享打印。当你通过因特网服务提供商（Internet Service Provider, ISP）把自己的网络连接到范围更大的网络

时, 你的网络便成了因特网的一部分。

因特网构建在一组协议之上, 这些协议涉及很多内容:

- 计算机如何寻址: 目前, 因特网上的每台计算机都有一个32位的数字与之关联——一个四字节的值, 写的时候通常像这样用小数点隔开: “101.132.64.15”。这些数字称为IP地址 (因特网协议地址)。

因特网中有一个域名系统, 通过它, 人们可以引用特定的计算机而无须知道其IP地址。比如, 在你访问http://www.cnn.com时, 实际访问的是http://157.166.226.26/。上一次我们直接尝试使用第二种写法时, 效果与第一种一样。但它有可能变化。网络中有一个域名服务器组成的网络维护着“www.cnn.com”这样的名字, 并将它们映射到“157.166.226.26”这样的地址。数字可以变化的事实是域名服务系统的一个优点——名字保持不变, 但可以指向任意地址。如果你连接的域名服务器坏了, 即使计算机仍连在网上也可能无法访问你想去的网站。直接输入IP地址倒是有可能访问到。

- 计算机如何通信: 通信数据被置于分组 (packet) 中, 分组具有明确定义的结构, 包括发送方地址、接收方地址, 以及每个分组的字节长度。
- 分组在网络中如何路由: 因特网是冷战时期设计的。它被设计成可以承受核攻击而保持通信不断。如果因特网的一部分被摧毁 (或损坏, 或因审查制度而遭封锁), 那么网络的路由机制会直接找到另一条路由来绕过损坏点。

对于网络中传输的数据, 其含义是由网络的最上层定义的。构建于因特网之上的第一个应用是电子邮件。经过这么多年的发展, 邮件协议已经变成了如今的邮局协议 (Post Office Protocol, POP) 和简单邮件传输协议 (Simple Mail Transfer Protocol, SMTP) 这样的标准。另一种古老而又重要的协议是文件传输协议 (File Transfer Protocol, FTP), 该协议支持在计算机之间传输文件。

这些协议并非复杂无比。通信结束时, 一台计算机通常会对另一台计算机说“BYE”或“QUIT”。而当一台计算机请求另一台计算机通过FTP接受文件时, 它会直接说“STO filename” (与前面一样, 早期的计算机开发者不愿多花两个字节来说“STORE”)。

万维网 (World Wide Web, WWW) 又是另外一套协议了, 它主要由Tim Berners-Lee开发。万维网构建于因特网之上, 只是在已有协议上又增加了更多的协议。

- 如何引用万维网上的东西: 万维网上的资源使用统一资源定位符 (uniform resource locator, URL) 来引用。URL指定了用于资源寻址的协议、提供资源的服务器域名, 以及资源在服务器上的路径。比如, 像http://www.cc.gatech.edu/index.html这样一个URL的含义是“使用HTTP协议与在www.cc.gatech.edu上的计算机通话, 并请求资源index.html”。

并非挂在因特网上的每一台计算机上的每一个文件都能通过URL来访问。首先, 可以从因特网访问的计算机上必须运行一套软件, 这套软件必须理解Web浏览器所理解的那种协议, 通常是HTTP或FTP。我们将运行这样一套软件的计算机称为服务器 (server)。访问服务器的浏览器则称为客户端 (client)。其次, 服务器上通常有可以访问的服务器目录。只有位于此目录及其子目录中的文件才能在万维网上通过URL访问到。

- 如何提供文档服务: 万维网上最常见的协议是HTTP。HTTP定义了网上的资源如何作为服务提供出来。HTTP真的很简单——浏览器可以直接对服务器说“GET index.html” (就是这些字母!)。

- 如何将文档格式化：万维网上的文档使用超文本标记语言（HyperText Markup Language, HTML）来格式化。

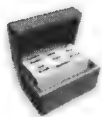
你会注意到，超文本这一术语在我们提及万维网的时候会频繁出现。超文本，正如其名，就是非线性文本。Ted Nelson发明了这一术语，用来描述Web上的这种在计算机出现之前不曾有过的阅读方式：在一个页面上阅读一小部分内容，点击一个链接，到另一个页面上再读一小部分，然后点击“返回”（Back）再回到原先的地方继续阅读。超文本的基本思想可以追溯到Vannevar Bush这个人，他是Franklin Roosevelt（富兰克林·罗斯福）总统的科学顾问之一，但直到计算机出现之后，人们才能够想象出如何实现Bush的麦麦克斯存储器（Memex）模型——一种捕捉思维流的设备。Tim Berners-Lee发明了万维网及其协议，在文档之间使用链接，并将它作为支持快速发布科研成果的方法。Web当然不是超文本系统的终极。像Ted Nelson研究的那种系统不会允许“死链接”（不再有效的链接）的出现。然而，尽管有各种瑕疵，Web是能够正常运行的。

浏览器（如Internet Explorer、Netscape Navigator、Mozilla和Opera）理解很多关于因特网的知识。通常，它知道多种协议，如HTTP、FTP、gopher（一种早期的超文本协议）和mailto（SMTP）。它还知道HTML如何格式化HTML文档，以及如何抓取HTML中引用的资源，如JPEG图片。访问因特网还可以不用耗费这么多开销。邮件客户端（例如Outlook和Eudora）理解上面提到的一部分协议，但并不理解全部。就连JES都知道一点SMTP和HTTP的知识，以支持作业提交。

与其他现代编程语言一样，Python也提供了支持因特网访问的模块，但开销比浏览器小得多。基本上，你可以用它来编写客户端小程序。Python的urllib模块支持打开一个URL并读取内容，就好像它们是文件一样。

```
>>> import urllib
>>> connection = urllib.urlopen("http://www.ajc.com/weather")
>>> weather = connection.read()
>>> connection.close()
```

使用这个例子，我们可以修正那个读取气温的程序（程序95），让它从因特网上直接读取气象页面。



程序98：从活动的气象页面上读取气温

```
def findTemperatureLive():
    # 获得气象页面
    import urllib # 也可以写到上面去
    connection = urllib.urlopen("http://www.ajc.com/weather")
    weather = connection.read()
    connection.close()
    # weatherFile = getMediaPath("ajc-weather.html")
    # file = open(weatherFile, "rt")
    # weather = file.read()
    # file.close()
    # 查找气温
    currLoc = weather.find("Currently")
    if currLoc <> -1:
        # 这时查找气温之后的"<b>&deg;"
```

```

temploc = weather.find("<b>&deg;", currLoc)
tempstart = weather.rfind(">", 0, temploc)
print "Current temperature: ", weather[tempstart + 1:temploc]
if currLoc == -1:
    print "They must have changed the page format -- can't find the temp"

```

程序原理

findTemperatureLive函数与之前的程序几乎完全一样，只是这次它直接从活动的AJC网站上读取字符串weather。我们使用urllib模块获得把Web页面读入字符串的能力，然后搜索这个字符串，搜索的方法与之前使用文件时完全一样。

我们可以通过Python的ftplib模块来使用FTP。

```

>>> import ftplib
>>> connect = ftplib.FTP("cleon.cc.gatech.edu")
>>> connect.login("guzdial","mypassword")
'230 User guzdial logged in.'
>>> connect.storbinary("STOR barbara.jpg",open(getMediaPath("barbara.jpg")))
'226 Transfer complete.'
>>> connect.storlines("STOR JESintro.txt",open("JESintro.txt"))
'226 Transfer complete.'
>>> connect.close()

```

要在Web上产生交互，我们需要能真正产生HTML的程序。比如，当你在一个文本输入框中输入一个词语，然后点击“搜索”（Search）按钮，实际上会使一个程序在服务器上运行起来，此程序执行你请求的搜索，并产生你所看到的HTML（Web页面）作为响应。Python语言经常用于这类程序的开发。它所具有的各种模块、灵活地使用引号的方式，以及易于编写的特点无不使之成为编写交互式Web程序的出色语言。

11.2 通过文本转换不同媒体

本章开头讲过，可以把文本看做单媒体。我们可以把声音映射成文本，反过来再把文本映射回声音，图片也一样。更有趣的是：可以把声音映射成文本……然后，把文本映射成图片。

可是，为什么要这样做呢？为什么要考虑用这种方式转换媒体呢？原因与我们将媒体数字化的原因是一样的。数字媒体转换成文本以后可以更方便地从一个地方传输到另一个地方，方便查错甚至纠错。事实上，当把二进制文件作为电子邮件的附件发送时，二进制文件首先会被转换成文本。一般来说，选择一种新的表示形式能使你实现新的功能。

将声音映射成文本很容易。声音只是一系列的样本（数字），把它们写到文件中方便得很。



程序99：将一段声音以文本数字的形式写入文件

```

def soundToText(sound,filename):
    file = open(filename,"wt")
    for s in getSamples(sound):
        file.write(str(getSampleValue(s))+ "\n")
    file.close()

```

程序原理

接受一段声音和一个文件名作为输入，然后以写文本（“wt”）方式打开文件。接下来，循

环遍历每一个样本并把它写入文件中。在这个程序中，使用了函数`str()`，用来把数字转换成相应的字符串表示，之后给它附上一个换行符并写到文件中。

用文本表示的声音对我们有什么用处呢？我们可以把它作为一系列数字来操作，就像Excel中那样（如图11.1所示）。这样，我们可以很方便地修改这些数字，比如将每个样本乘以2。我们甚至可以根据这些数字来绘图，然后就像在MediaTools应用程序中那样查看声音图形（如图11.2所示）。（但这里会出现一个错误——Excel的绘图功能不喜欢点的数目超过32 000个，而基于每秒22 000个样本的采样速率，32 000个样本发不出太多声音。）

怎样将这一系列数字转回声音呢？假设你修改了Excel中的某些数字，现在想听听修改后的结果，怎样才能做到呢？Excel这边很简单：把那列数字复制到一份新的工作表中，存为文本，然后取得文本文件的路径名，在Python程序中使用即可。

程序本身则复杂一些。声音转换成文本的时候，可以用`getSamples`取得所有样本并输出。但现在我们何以知晓文件中有多少行呢？我们不能使用`getLines`——没有这个函数。需要提防的问题有两个：（a）文件的行数多于先前读取的声音样本数目；（b）到达声音对象结尾之前已经用光了所有的行。

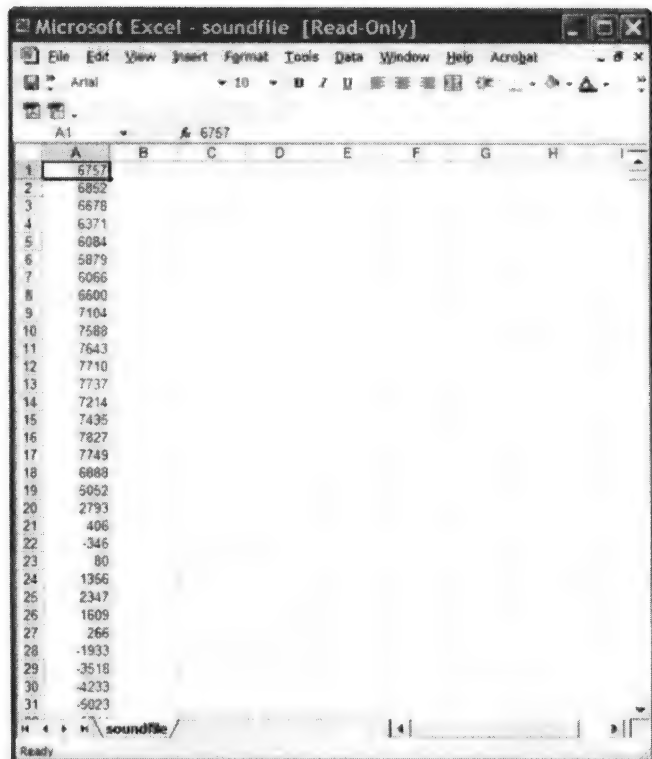


图11.1 声音转换成文本文件后读入Excel中

我们打算用一个`while`循环来实现目标。在前面的章节中，我们曾简单地使用过`while`循环，它就像`if`，接受一个表达式并在表达式取得真值时执行后面的语句块。它与`if`的区别在于，执行语句块之后，`while`循环会再次检查其表达式。如果表达式仍然为真，整个语句块会再次执行。最终你会期望表达式变为假，这时，`while`语句块之后的一行将会执行。如果不是这样，那你就得到了一个无限循环——循环永远持续下去（理论上）。

```
while 1==1:
    print "This will keep printing until the computer is turned off."
```

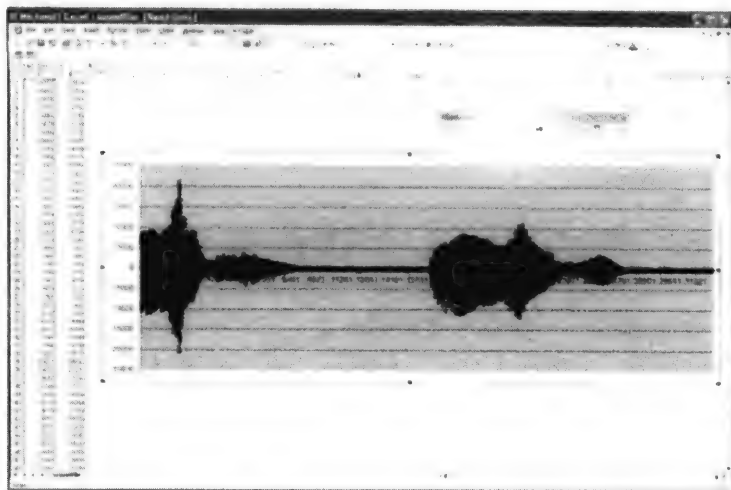


图11.2 基于文本表示的声音在Excel中作图

文本转换成声音的例子我们打算这样实现:只要文件中还有数字而且声音对象中还有空间,那么我们就不断从文件中读取样本并保存到声音中。我们用`float()`函数将数字字符串转换成实数。(用`int`也可以,但我们想找个机会介绍一下`float`。)

```
>>> print 2*"123"
123123
>>> print 2 * float("123")
246.0
```



程序100: 将文件中的数字文本转换成声音

```
def textToSound(filename):
    # 设置声音对象
    sound = makeSound(getMediaPath("sec3silence.wav"))
    soundIndex = 0
    # 设置文件
    file = open(filename, "rt")
    contents = file.readlines()
    file.close()
    fileIndex = 0
    # 一直循环,直到声音对象的空间用完或文件内容用完
    while (soundIndex < getLength(sound)) and (fileIndex < len(contents)):
        sample = float(contents[fileIndex])
        # 取得文件的一行
        setSampleValueAt(sound, soundIndex, sample)
        fileIndex = fileIndex + 1
        soundIndex = soundIndex + 1
    return sound
```

程序原理

`textToSound`函数接受一个文件名作为输入，文件中包含用数字表示的样本。打开一个3秒长的静音文件来保存声音。`soundIndex`代表要写入的下一个样本，而`fileIndex`代表要从文件内容列表`contents`中读取的下一个数字。`while`循环语句的意思是：不断进行直到`soundIndex`越过了声音的长度或者`fileIndex`越过了文件的末尾（文件内容保存在`contents`列表中）。在一次循环中，将列表中的下一个字符串转换成浮点数，把样本值设为该浮点数，然后同步递增`fileIndex`和`soundIndex`。循环结束的时候（任一个条件不再为真），返回了新建的声音。

实际上，声音映射成文本之后，我们不一定再把它映射回声音。可以考虑以图片为目标。下面的程序接受一段声音并将每一个样本映射成一个像素。只需定义自己想要的映射方式：即如何表示这些样本。我们选用了一种非常简单的方法：大于1 000的样本对应红色像素，小于-1 000的样本对应蓝色像素，其他样本全部对应绿色像素（如图11.3所示）。

现在，我们必须处理用完像素之前取完样本的情况。为解决这个问题，我们使用了另外一种编程设施：`break`。`break`语句终止当前的循环并转向循环之后的语句。在这个例子中，如果样本用完，我们便终止处理像素的`for`循环。



程序101：将声音视觉化

```
def soundToPicture(sound):
    picture = makePicture(getMediaPath("640x480.jpg"))
    soundIndex = 0
    for p in getPixels(picture):
        if soundIndex > getLength(sound):
            break
        sample = getSampleValueAt(sound, soundIndex)
        if sample > 1000:
            setColor(p, red)
        if sample < -1000:
            setColor(p, blue)
        if sample <= 1000 and sample >= -1000:
            setColor(p, green)
        soundIndex = soundIndex + 1
    return picture
```

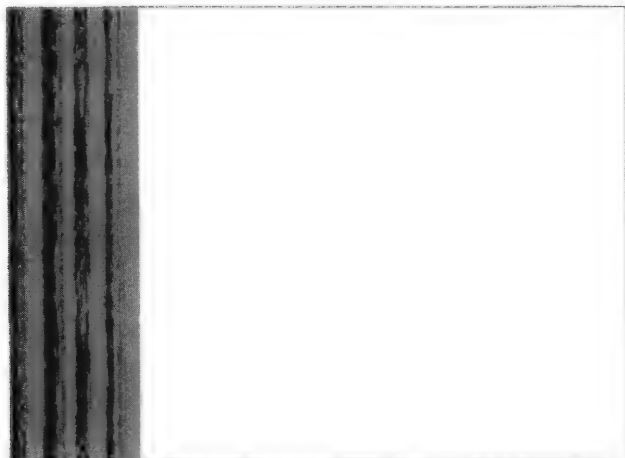


图11.3 声音“*This is a test*”的视觉效果

程序原理

在soundToPicture函数中,我们接受一段声音作为输入并打开了一幅640×480的空白图片。针对图片中的每个像素,我们在soundIndex处取得一个样本值,计算它应该映射成哪种颜色,并将像素p设成那种颜色。然后递增soundIndex。如果soundIndex越过了声音末尾,那么就使用break语句从循环中跳出来。最后返回了新建的图片。

考虑一下WinAmp是如何做声音视觉化的,Excel和MediaTools是如何作图的,上面的程序又是如何做视觉化的。每个程序所做的只是确定了一种从样本映射到颜色和空间的方法。一切只是映射,就是这样。



计算机科学思想:这一切都是比特

声音、图片和文本都只是“位”而已。它们只是信息。我们可以把一种媒体映射成任何其他媒体,甚至再转换回原先的媒体。只需要定义一种表示方式。

使用列表作为媒体表示的结构化文本

我们已经说过,列表(list)是非常强大的。将声音变成列表非常容易。



程序102: 将声音映射成列表

```
def soundToList(sound):
    list = []
    for s in getSamples(sound):
        list = list + [getSampleValue(s)]
    return list

>>> list = soundToList(sound)
>>> print list[0]
6757
>>> print list[1]
6852
>>> print list[0:100]

[6757, 6852, 6678, 6371, 6084, 5879, 6066, 6600,
7104, 7588, 7643, 7710, 7737, 7214, 7435, 7827,
7749, 6888, 5052, 2793, 406, -346, 80, 1356, 2347,
1609, 266, -1933, -3518, -4233, -5023, -5744,
-7394, -9255, -10421, -10605, -9692, -8786, -8198,
-8133, -8679, -9092, -9278, -9291, -9502, -9680,
-9348, -8394, -6552, -4137, -1878, -101, 866, 1540,
2459, 3340, 4343, 4821, 4676, 4211, 3731, 4359, 5653,
7176, 8411, 8569, 8131, 7167, 6150, 5204, 3951, 2482,
818, -394, -901, -784, -541, -764, -1342, -2491,
-3569, -4255, -4971, -5892, -7306, -8691, -9534,
-9429, -8289, -6811, -5386, -4454, -4079, -3841,
-3603, -3353, -3296, -3323, -3099, -2360]
```

将图片变成列表也同样容易——同样只需要定义一种表示。可不可以将每个像素先按X和Y坐标,再按红、绿、蓝通道来映射呢?我们必须使用双重方括号,因为我们想用大列表中套子列表的形式来表示这5个值。



程序103：将图片映射成列表

```
def pictureToList(picture):
    list = []
    for p in getPixels(picture):
        list = list + [[getX(p),getY(p),getRed(p),getGreen(p),getBlue(p)]]
    return list

>>> picture = makePicture(pickAFile())
>>> piclist = pictureToList(picture)
>>> print piclist[0:5]
[[1, 1, 168, 131, 105], [1, 2, 168, 131, 105], [1, 3, 169,
132, 106], [1, 4, 169, 132, 106], [1, 5, 170, 133, 107]]
```

再转换回去也不难。只需确保X和Y坐标位于画布的边界之内。



程序104：将列表映射成图片

```
def listToPicture(list):
    picture = makePicture(getMediaPath("640x480.jpg"))
    for p in list:
        if p[0] <= getWidth(picture) and p[1] <= getHeight(picture):
            setColor(getPixel(picture,p[0],p[1]),makeColor(p[2],p[3],p[4]))
    return picture
```

可以想象上面的程序是有效的，因为我们能看出这个映射是双向的，在这里只考虑列表到图片的映射就可以了。数字不一定非要来自图片，我们可以同样方便地将气象数据、股票行情数据或其他任何数据映射成一个数字列表，然后可视化。一切都是位……

在这个程序中，我们真正完成的工作只是改变编码方式，基本的信息根本没有变。不同的编码方式提供了不同的能力。

一位绝顶聪明的数学家，Kurt Gödel，利用编码的概念完成了20世纪最伟大的证明。他证明了不完备性定理（incompleteness theorem），从而证明了任何强大的数学系统都不能证明所有的数学真理（mathematical truth）。他设计了一种将真理的数学陈述映射为数字的方式。这个设计远早于ASCII码的出现，那时像这样的映射还不像后来那么司空见惯。一旦这些陈述成了数字，他便能证明某些代表真陈述的数字是无法基于数学系统来导出的。通过这种方法，他证明了没有任何一种逻辑系统可以证明所有的真陈述。使用编码变换，他获得了新的能力，从而证明出了以前没有人知道的东西。

Claude Shannon是美国的一位工程师兼数学家，他发展了信息论（information theory）。信息论描述了信息可以怎样在不同的媒体中表示。当我们把一幅图片映射成文本和声音的时候，实际就是在应用信息论。

11.3 在图片中隐藏信息

信息隐藏术（steganography）是使用不易察觉的方法来隐藏信息的技术。如果有一条文本消息使用黑色字体显示在白色图片中，我们可以把它隐藏进另一幅图片中。为实现这一目标，首先我们可以把新图片中的红色全部变成偶数，然后循环遍历包含待隐藏文本的图片像素，如果像素的颜色接近黑色，那么就在新图片中把相应像素的红色值改为奇数。



程序105：编码消息

```
def encode(msgPic, original):
    # 假定msgPic和original具有相同的尺寸
    # 首先把所有的红色值变成偶数
    for px1 in getPixels(original):
        # 使用模运算测试奇偶性
        if(getRed(px1) % 2) == 1:
            setRed(px1, getRed(px1) - 1)
    # 然后针对msgPic中所有的黑色像素
    # 把original中相应像素的红色值改为奇数
    for x in range(0, getWidth(original)):
        for y in range(0, getHeight(original)):
            msgPx1 = getPixel(msgPic, x, y)
            origPx1 = getPixel(original, x, y)
            if (distance(getColor(msgPx1), black) < 100.0):
                # 这是一个消息像素！把红色值改为奇数
                setRed(origPx1, getRed(origPx1) + 1)
```

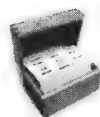
现在可以通过如下命令在沙滩图片中隐藏信息。

```
>>> beach = makePicture(getMediaPath("beach.jpg"))
>>> explore(beach)
>>> msg = makePicture(getMediaPath("msg.jpg"))
>>> encode(msg, beach)
>>> explore(beach)
>>> writePictureTo(beach, getMediaPath("beachHidden.png"))
```

保存图片时应该使用png或bmp格式，不要使用JPGE（jpg）格式。JPEG标准是有损（lossy）的，也就是说它不会精确地按图片的本来面目来保存，而会丢弃一些细节（比如特定的红色值），这些细节你通常不会注意到，但在这种情形中我们想精确地保存图片的本来面目，从而以后可以将消息解码出来。

你能看出原来的沙滩图片和隐藏了信息的图片之间有区别吗（如图11.4所示）？如果能，我们表示怀疑。

现在我们把隐藏的信息取回来。下面是完成这项任务的函数。



程序106：解码消息

```
def decode(encodedImg):
    # 接受一幅编码了消息的图片，返回原来的消息
    message = makeEmptyPicture(getWidth(encodedImg), getHeight(encodedImg))
    for x in range(0, getWidth(encodedImg)):
        for y in range(0, getHeight(encodedImg)):
            encPx1 = getPixel(encodedImg, x, y)
            msgPx1 = getPixel(message, x, y)
            if (getRed(encPx1) % 2) == 1:
                setColor(msgPx1, black)
    return message
```



图11.4 原来的图片（左）和隐藏了信息的图片（右）

我们可以用下面的代码来解码消息。这次你应该能读出结果中的消息了（如图11.5所示）。

```
>>> origMsg = decode(beach)
>>> explore(origMsg)
```



图11.5 解码之后的消息

编程摘要

通用程序片段

<code>while</code>	创建一个循环，只要提供给 <code>while</code> 语句的逻辑表达式为真（即非0）便一直迭代执行循环体
<code>break</code>	立即打断一个循环——跳转到 <code>while</code> 或 <code>for</code> 循环的末尾
<code>urllib, ftplib</code>	用于访问URL或FTP的模块
<code>str</code>	将数字（或其他对象）转换成相应的字符串表示
<code>float</code>	将数字或字符串转换成等值的浮点数表示

习题

- 11.1 找一张包含大量文本的网页，比如<http://www.cnn.com>，使用浏览器的相关菜单项将它另存为文件，比如MYPAGE.HTML，然后使用JES甚至Windows记事本那样的编辑器来编辑它。找出一些浏览页面时可以看到文本，比如标题或文章内容。然后修改它！比如把“抗议者骚乱”中的“抗议者”改成“大学生”甚至“幼儿园教师”。现在，在浏览器中重新打开页面。看到了没有，你刚刚改写了新闻！
- 11.2 创建一个函数，从多个网站获取数据，然后将它们整合到同一张Web页面中。
- 11.3 创建一个函数，从某个URL处获取一段声音，然后使用它制作一段声音剪辑并保存在本地机器上。
- 11.4 创建一个函数，从某个URL处获取一幅图片，为它制作一张缩略图并保存在本地机器上。
- 11.5 将下列词语跟后面的定义对应起来，把定义前面的字母填写到词组左边的横线上。（是的，会有一个定义用不到。）
- _____域名服务器 _____Web服务器 _____HTTP _____HTML
 _____客户端 _____IP地址 _____FTP _____URL
- (a) 一台计算机，能将www.cnn.com这样的名字匹配成相应的因特网地址。
- (b) 一种协议，用于在计算机之间移动文件（比如，将文件从你的个人计算机移动到一台更大的充当Web服务器的计算机上）。
- (c) 一个字符串，解释在因特网上可以从哪台计算机（域名）上以何种方式（协议）从哪个位置（路径）找到某个特定文件。
- (d) 通过HTTP提供文件的计算机。
- (e) 一种协议，多数网站都基于它来构建，形式简单，以快速传输少量信息为设计目标。
- (f) 当浏览器（比如Internet Explorer）联系yahoo.com这样的服务器时，它的角色是什么？
- (g) Web页面中的标签，用于确定页面的不同部分及其格式化方式。
- (h) 在计算机之间传输电子邮件的协议。
- (i) 计算机在因特网上的数字标志——由四个0~255之间的数字组成，如120.32.189.12。
- 11.6 针对下面每种实体，看自己能否基于位或字节想象出它的表示方式。
- (a) 因特网地址由四个0~255之间的数字组成。一个因特网地址有多少位？
- (b) Basic编程语言中可以用数字为代码编行号，数字的范围在0~65 535之间。这样一个行号需要多少位来表示？

- (c) 每个像素的颜色值有三个分量：红、绿和蓝，每个分量的范围在0~255之间。表示一个像素的颜色需要多少位？
- (d) 某些系统中字符串的最大长度为1 024个字符。表示这样一种字符串的长度时需要多少位？

11.7 域名服务器是什么？它完成什么功能？

11.8 FTP、SMTP和HTTP分别是什么？各自用来做什么？

11.9 超文本（HyperText）是什么？它是由谁发明的？

11.10 客户端和服务器的区别是什么？

11.11 懂得如何处理文本对你在因特网上收集和生成信息有何帮助？

11.12 因特网（Internet）是什么？

11.13 ISP是什么？你能给出一个ISP的例子吗？

11.14 编写一个函数将图片映射成一段声音。

11.15 有没有可能将一幅彩色图片隐藏在另一幅图片之中？为什么？

11.16 编写一个函数将文本翻译成一幅图片。

11.17 编写一个函数将文本翻译成一段声音。

11.18 编写一个函数，通过将每个字母替换成字母表中它后面的一个字母来加密文本。同时编写一个函数来解密这样的消息。

11.19 编写一个函数，通过改变消息中各个字符的次序来加密文本。同时编写一个函数来解密这样的消息。

11.20 编写一个函数，使用各个字母在另一篇文档中的位置来加密文本。同时编写一个函数来解密这样的消息。

深入学习

Mark用来趟过Python模块之河的一本书是Frederik Lundh的《Python Standard Library》（Python标准库）[29]。另外，可以从如下网址找到库模块的列表及其文档：<http://docs.python.org/library/>。

产生Web文本

本章学习目标

本章媒体学习目标:

- 掌握使用HTML的基本技巧。
- 根据输入数据自动产生HTML，比如为图片目录产生索引页面。
- 使用数据库产生Web内容。

本章计算机科学学习目标:

- 用另一种计数基数：十六进制，来指定RGB颜色。
- 区分XML和HTML。
- 解释SQL是什么以及它与关系型数据库的关系。
- 创建和使用子函数（功能函数）。
- 演示散列表（字典）的一种用法。

12.1 HTML：Web的表示方法

万维网以文本为主要载体，其中的文本又主要以超文本标记语言（HyperText Markup Language, HTML）格式来编码。HTML是在标准通用标记语言（Standard Generalized Markup Language, SGML）标准的基础上发展起来的，SGML通过向文本中增添附加的文本来标记文档中不同的逻辑部分：“这里是题目”、“这里是标题”、“这里只是一段有序列表”。早期的HTML（与SGML一样）只是为标记文档的不同部分，文档的外观取决于浏览器。人们能想到文档在一种浏览器中的外观与另一种浏览器中不同。但随着Web的演化，两个独立的目标形成了：描述大量逻辑部分的能力（比如描述价格、部件编号、股票代码、气温等）和精细的格式化控制能力。

针对第一个目标，可扩展标记语言（eXtensible Markup Language, XML）应运而生。它允许定义新的标签，如<partnumber>7834JK</partnumber>。针对第二个目标，像层叠样式表（Cascading Style Sheets, CSS）之类的技术发展了起来。此外，还有另一种标记语言，XHTML，也发展了起来，它是基于XML的HTML。

本章以介绍XHTML为主，但不打算把它与原始的HTML进行区分，而直接把它作为HTML来讨论。

也不打算在这里提供完整的HTML教程。这样的教程已经有很多了，印刷品和网上的在线教程都有，其中有很多质量也不错。使用自己常用的搜索引擎搜一下“HTML tutorial”（HTML学习指南），挑一种喜欢的即可。这里要讨论的是HTML的一般概念，顺便提及你必须知道的一些标签。

标记语言的含义就是在原始文本中插入别的文本来标志不同的部分。在HTML中，插入的文本（称为标签）使用尖括号（小于号和大于号）来分隔。比如，<p>开启一个段落，而</p>结束一个段落。

Web页面的内容可分为不同部分，且部分之间可以相互嵌套。首先，页面的顶部会有一个doctype，声明页面的种类（即浏览器应当把它作为HTML、XHTML、CSS，还是其他的东西来解释）。doctype之后是页面头部（<head>...</head>）和页面主体（<body>...</body>）。头部可以嵌入题目（title）之类的信息——题目在头部之前结束。主体中可以嵌入很多内容，比如标题（h1在主体结束之前开始并结束）和段落。页面头部和页面主体全部嵌在一对（<html>...</html>）标签中。图12.1显示了一个简单的页面源文件，图12.2显示了该页面在Internet Explorer中的显示效果。你可以尝试一下，只需要在JES中输入页面内容并另存为一个名字带有HTML后缀的文件，然后在Web浏览器中打开它。这个文件与Web页面的唯一区别在于它存在于磁盘上。如果它位于某个Web服务器上，那就是Web页面了。



常见bug：浏览器是宽容的，但通常会有错

浏览器非常宽容。如果你忘了写DOCTYPE或者在HTML中写错了东西，那么浏览器只会猜测你的意图并尽量显示它。然而，墨菲法则（Murphy's Law）告诉我们它会猜错。（墨菲法则：有可能出错的事情总是要出错的（Anything that can go wrong will go wrong）。参阅：http://en.wikipedia.org/wiki/Murphy%27s_law。——译者注）如果想让Web页面完全显示成你想要的样子，还是应该把HTML写对。

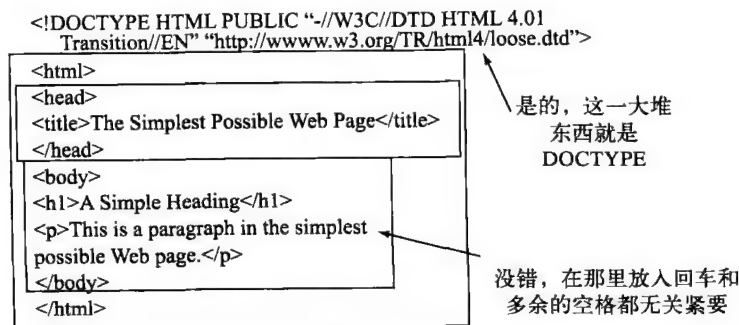


图12.1 简单的HTML页面源文件

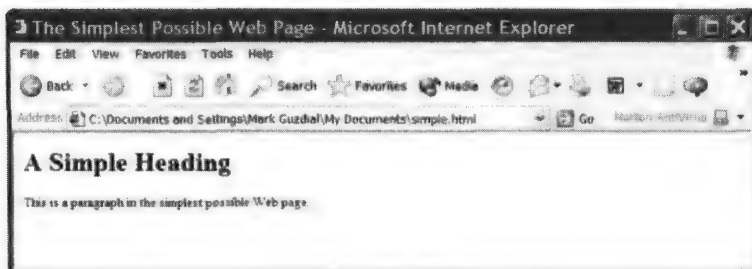


图12.2 在Internet Explorer中打开简单的HTML页面

下面的一些标签是你应当熟知的：

- <body>标签可接受设置背景、文本和链接颜色的参数。这些颜色可以是简单的颜色名字，比如“red”或“green”，也可以是特定的RGB颜色值。

指定颜色时，可以用十六进制数。十六进制也是一种计数系统，正如十进制系统的基数为10，十六进制计数使用16做基数。十进制数1到20翻译成十六进制依次是1、2、3、

4、5、6、7、8、9、A、B、C、D、E、F、10、11、12、13和14。十六进制数“14”可以看成由一个16和四个1组成，所以结果是20。

十六进制的优点在于每个数字对应4位，两个十六进制数字正好对应一个字节。这样，三字节的RGB颜色就可以用6位十六进制数以R、G、B的顺序表示。十六进制数FF0000是红色——红色分量为255（FF），绿色分量为0，蓝色分量为0。0000FF是蓝色，000000是黑色，FFFFFF是白色。

- 标题使用标签<h1>...</h1>到<h6>...</h6>表示。数字越小，标题越醒目。
- 还有大量标签用于其他样式：表示强调的...，表示斜体的<i>...</i>，表示粗体的...，更大字体的<big>...</big>和更小字体的<small>...</small>，打字机字体<tt>...</tt>，预格式化文本<pre>...</pre>，块引用<blockquote>...</blockquote>，下标_{...}和上标^{...}（如图12.3所示）。另外还可以用...这类标签来控制字体和颜色这类属性。
- 可以用
插入断行而不必另起新的段落。
- 可以用<image src="image.jpg"/>这样的标签插入图片（如图12.4所示）。image标签以src=参数的形式接受一个图片参数。src=之后是图片的规格说明，其形式可以有以下几种：
 - 如果只是一个文件名（如"flower1.jpg"），则认为它是一幅图像，且位于引用它的HTML所在的目录中。
 - 如果是一条路径，则假定它是一条从HTML页面所在目录开始的路径。所以，如果“My Documents”目录下有一张HTML页面，它引用了mediasources目录下的图片，则页面中可以引用"mediasources/flower1.jpg"。这种情况下，你可以使用UNIX中的惯例（比如用“..”引用父目录，于是“../images/flower1.jpg”的含义是：回到父目录，然后下到Images目录中去取flower1.jpg）。

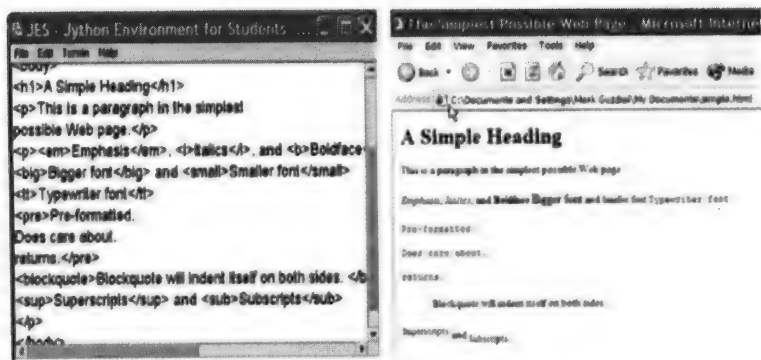


图12.3 HTML样式

- 还可以是一个完整的URL——完全可以引用其他服务器上的图片。
可以用image标签的选项来控制图像的宽度和高度（比如使用<image height="100" src="flower.jpg">将图像高度限制在100个像素），或者在维持高度/宽度比不变的前提下调整宽度。另外，还可以用alt选项指定图片无法显示时可代替之的文本（比如在音频浏览器或布莱叶盲文浏览器中）。
- 可以用锚（anchor）标签anchor text创建指向别处的链接“anchor text”。在这个例子中，someplace.html是锚的目标——单击“anchor

text”时浏览器便会转到这个位置。锚就是你点击的东西，它可以是“anchor text”这样的文本，也可以是图片。如图12.5所示，目标也可以是完整的URL。

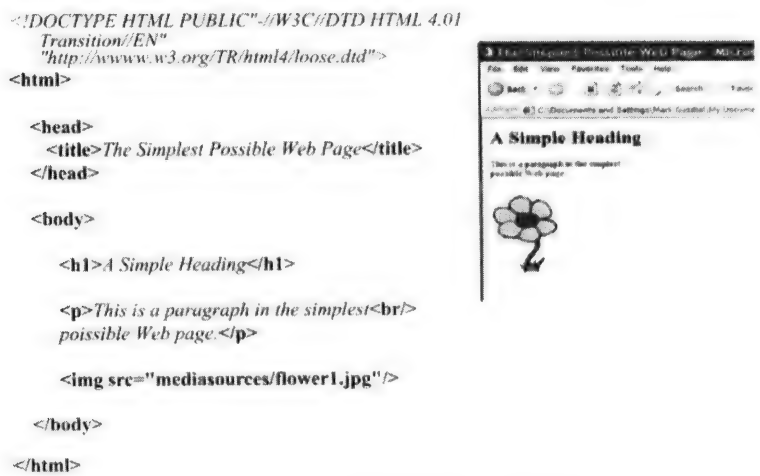


图12.4 在HTML页面中插入图像

- 还要注意，图12.5中源文件中的断行在浏览器中并没显示出来。断行甚至可以出现在锚标签的中间而不影响显示结果。影响结果的断行（即可以从浏览器中看到的）要用 `
` 或 `<p>` 标签来产生。
- `...` 和 `...` 标签分别产生项目符号列表（无序列表）和编号列表（有序列表）。列表中的各项目使用 `...` 来指定。
- 表格使用 `<table>...</table>` 标签来创建。表格由表格行组成，表格行使用 `<tr>...</tr>` 标签创建，每一行由多个表格数据项组成，每一项使用 `<td>...</td>` 来标志（如图12.6所示）。表格包含行，行包含表格数据项。

关于HTML的知识还有很多，比如框架（HTML页面窗口中的子窗口）、区域（`<div />`）、水平条（`<hr />`）以及applet和JavaScript。对于本章后续内容的理解，以上的项目是最关键的。

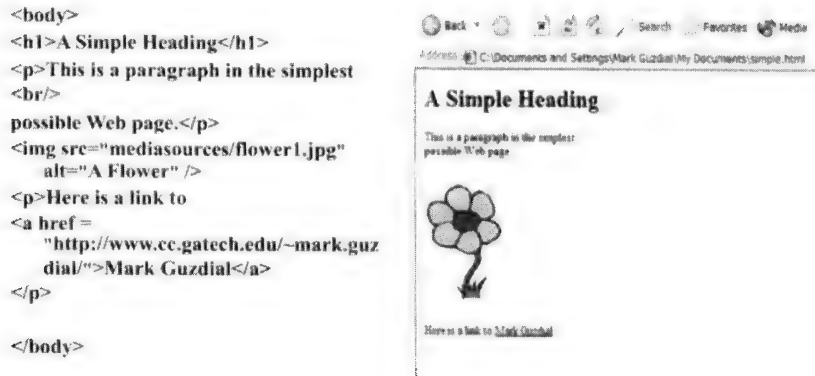


图12.5 含有链接的HTML页面

```

<table border="5">
<tr><td>Column
1</td><td>Column
2</td></tr>
<tr><td>Element in column
1</td><td>Element in
column 2</td></tr>
</table>

```

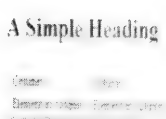


图12.6 在HTML页面中插入表格

12.2 编写程序产生HTML

HTML本身不是一种编程语言。HTML不能指定循环、条件、变量、数据类型，或任何其他我们学过的用于描述过程的东西。HTML描述的是结构，而非过程。

然而，编写程序来产生HTML并不难。Python定义字符串时的多种引号方式这时就真正派上用场了！



程序107：产生简单的HTML页面

```

def makePage():
    file=open("generated.html","wt")
    file.write("""<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transition//EN" "http://www.w3.org/TR/html4/loose.dtd">

<html>
<head> <title>The Simplest Possible Web Page</title>
</head>
<body>
<h1>A Simple Heading</h1>
<p>Some simple text.</p>
</body>
</html>""")
    file.close()

```

程序没有问题，但这样的写法让人厌倦。为什么要编写一个程序来输出本来可以用文本编辑器直接编辑的内容呢？编写程序应该是为了可复制，为了传达过程和裁剪功能。我们还是做一个主页产生器吧。



程序108：第一个主页产生器

```

def makeHomePage(name, interest):
    file=open("homepage.html","wt")
    file.write("""<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transition//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>"""+name+"""'s Home Page</title>
</head>
<body>
<h1>Welcome to """+name+"""'s Home Page</h1>
<p>Hi! I am """+name+"". This is my home page!
I am interested in """+interest+"""</p>
</body>
</html>""")
    file.close()

```

有了上面的程序，执行makeHomePage("Barb", "horses")命令将产生如下页面：

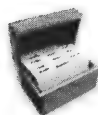
```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transition//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
<title>Barb's Home Page</title>
</head>
<body>
<h1>Welcome to Barb's Home Page</h1>
<p>Hi! I am Barb. This is my home page!
I am interested in horses</p>
</body>
</html>
```



调试技巧：先编写出HTML

产生HTML的程序看上去总是扑朔迷离的。在开始这样的程序之前，可以先把HTML写出来。根据自己想要的样子来组织一个HTML的例子，确保它能正常显示在浏览器中。然后再编写产生这种HTML的函数。

要修改这个程序还是很痛苦。HTML有太多细节，各种引号也不好对付。最好还是用子函数把上面的程序分解成更容易使用的片段。这又是使用过程式抽象的一个例子。在下面的程序版本中，我们把前面最有可能修改的部分精简成了函数调用。



程序109：改进的主页产生器

```
def makeHomePage(name, interest):
    file=open("homepage.html","wt")
    file.write(doctype())
    file.write(title(name+'s Home Page'))
    file.write(body("""
<h1>Welcome to """+name+"""'s Home Page</h1> <p>Hi! I am
"""+name+""". This is my home page! I am interested in
"""+interest+"""</p>"""))
    file.close()

def doctype():
    return '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transition//EN" "http://www.w3.org/TR/html4/loose.dtd">'

def title(titlestring):
    return "<html><head><title>"+titlestring+"</title></head>"

def body(bodystring):
    return "<body>"+bodystring+"</body></html>"
```

我们可以从任何地方抓取Web页面所需的内容。下面的菜谱可以从输入参数提供的目录中取得信息并为目录中的图片创建索引页面（如图12.7所示）。我们打算在这里列出doctype()和其他功能函数——我们把精力集中在自己关心的部分上。这也是我们应当采取的考虑问题的方式——只考虑自己关心的部分；doctype()我们已经写过一次了，那就忘了它吧！

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD
HTML 4.01 Transition//EN"
"http://www.w3.org/TR/html4/loose.dtd"
><html><head><title>Samples from
C:\Documents and Settings\Mark
Guzdial\My
Documents\mediasources\pics</title></hea
d><body><h1>Samples from
C:\Documents and Settings\Mark
Guzdial\My Documents\mediasources\pics
</h1>
<p>Filename: students1.jpg<image
src="students1.jpg" height="100" /></p>
<p>Filename: students2.jpg<image
src="students2.jpg" height="100" /></p>
<p>Filename: students5.jpg<image
src="students5.jpg" height="100" /></p>
<p>Filename: students6.jpg<image
src="students6.jpg" height="100" /></p>
<p>Filename: students7.jpg<image
src="students7.jpg" height="100" /></p>
<p>Filename: students8.jpg<image
src="students8.jpg" height="100" /></p>
</body></html>
```

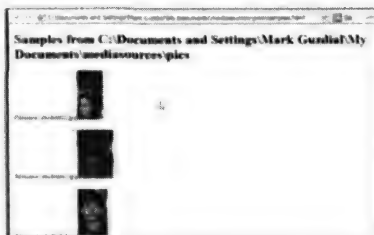


图12.7 创建缩略图页面



程序110：产生一张缩略图页面

```
import os

def makeSamplePage(directory):
    samplesFile=open(directory+"//samples.html","wt")
    samplesFile.write(doctype())
    samplesFile.write(title("Samples from "+directory))
    # 现在，我们组织页面的body字符串
    samples="<h1>Samples from "+directory+" </h1>"
    for file in os.listdir(directory):
        if file.endswith(".jpg"):
            samples=samples+"<p>Filename: "+file
            samples=samples+'<image src="'+file+'" height="100"/></p>'
    samplesFile.write(body(samples))
    samplesFile.close()
```

我们还可以从Web上抓取信息来创建新的Web内容。*Google News*[⊖]之类的网站就是这么做的。下面的主页产生器版本会动态地收集气温信息（专门针对亚特兰大市）。



程序111：产生包含气温信息的主页

```
import urllib

def makeHomePage(name, interest):
    file=open("homepage.html","wt")
    file.write(doctype())
    file.write(title(name+"'s Home Page"))
    file.write(body("""
<h1>Welcome to ""'+name+""'s Home Page</h1> <p>Hi! I am
""'+name+""'. This is my home page! I am interested in
""'+interest+""'</p> <p>Right here and right now it's
""'+findTemperatureLive()+""' degrees. (If you're in
the North, nyah-nyah!)""'))
    file.close()
```

⊖ <http://news.google.com>。

```
def findTemperatureLive():
    # 获得气象页面
    import urllib
    connection=urllib.urlopen("http://www.ajc.com/weather")
    weather = connection.read()
    connection.close()
    #weatherFile = getMediaPath("ajc-weather.html")
    #file = open(weatherFile,"rt")
    #weather = file.read()
    #file.close()
    # 查找气温
    curLoc = weather.find("Currently")
    if curLoc <> -1:
        # 这时查找气温之后的"<b>&deg;"
        tempLoc = weather.find("<b>&deg;",curLoc)
        tempStart = weather.rfind(">",0,tempLoc)
        return weather[tempStart+1:tempLoc]
    if curLoc == -1:
        return "They must have changed the page format--can't find the temp"
```

还记得那个随机产生句子的程序吗？我们也可以把它加到主页上去。



程序112：带有随机语句的主页产生器

```
import urllib
import random

def makeHomePage(name, interest):
    file=open("homepage.html","wt")
    file.write(doctype())
    file.write(title(name+"'"s Home Page"))
    file.write(body("""
<h1>Welcome to ""'+name+""'"s Home Page</h1> <p>Hi! I am
""'+name+"""". This is my home page! I am interested in
""'+interest+"""/> <p>Right here and right now it's
""'+findTemperatureLive()+"""/> degrees. (If you're in the
North, nyah-nyah!).</p> <p>Random thought for the day:
""'+sentence()+"""/>"))
    file.close()

def sentence():
    nouns = ["Mark", "Alicia", "Maria", "Latrice", "Jose", "Corey", "Tashaun"]
    verbs = ["runs", "skips", "sings", "leaps", "jumps", "climbs", "argues", "giggles"]
    phrases = ["in a tree", "over a log", "very loudly", "around the-bush", "while reading the news"]
    phrases = phrases + ["very badly", "while skipping", "instead of-grading", "while typing on the CoWeb."]
    return random.choice(nouns)+" "+random.choice(verbs)+" "+random.choice(phrases)+". "

def findTemperatureLive():
    # 获得气象页面
    import urllib
    connection=urllib.urlopen("http://www.ajc.com/weather")
    weather = connection.read()
    connection.close()
    #weatherFile = getMediaPath("ajc-weather.html")
```

```

#file = open(weatherFile,"rt")
#weather = file.read()
#file.close()
# 查找气温
curLoc = weather.find("Currently")
if curLoc <> -1:
    # 这时查找气温之后的"<b>&deg;"
    tempLoc = weather.find("<b>&deg;", curLoc)
    tempStart = weather.rfind(">", 0, tempLoc)
    return weather[tempStart+1:tempLoc]
if curLoc == -1:
    return "They must have changed the page format--can't find the temp"

```

~这几行程序应该与下面的一行连接。Python中一条命令不可以跨多行。

程序原理

让我们一步步解释这个巨大的示例程序：

- 这个函数需要urllib和random，因此在程序开头导入（import）了这两个模块。
- 主函数是makeHomePage。调用时，传入一个名字（name）和页面中将会提及的一项兴趣（interest）。
- 在makeHomePage开始的地方，打开了将要输出的HTML文件，然后使用先前的功能函数编写了doctype（这里没有列出它，但它必须位于程序区中）。我们又输出了标题（使用title函数），其间插入了函数参数中的name，接着我们输出了页面主体（使用body函数）。
- body函数的输入是个长长的字符串，它实际上是通过调用findTemperatureLive()和sentence()这两个函数构造出来的。我们使用三重引号，这样可以在字符串中随意输入回车。我们把name和interest连接到了HTML字符串中，从而这些信息也输出到了结果中。
- 在构造body字符串的过程中，我们调用了findTemperatureLive()。注意，这次的程序与我们上一章的版本有所不同。这一次我们返回了结果字符串，而不是用print直接输出。返回字符串，我们便可以使用返回的结果并把它插入到HTML的主体字符串中去。
- 我们调用了sentence()，把它作为“Random thought for the day:”的内容。与findTemperatureLive()一样，sentence()跟之前的版本也有不同：使用return返回结果而不用print打印结果。这样，我们可以把它也输出到HTML的主体字符串中。
- 最后，再回到makeHomePage，我们关闭了HTML文件，任务完成。

你认为大型Web站点都是从哪里取得信息的呢？这些网站拥有的页面数量巨大。这些页面都来自哪里，又存放于哪里呢？

12.3 数据库：存放文本的地方

大型Web站点使用数据库（database）来存储文本和其他信息。像EBay.com、Amazon.com和CNN.com这样的网站都拥有存储巨量信息的大型数据库。这些站点的页面并不是某人输入信息来产生的。它们是用程序访问数据库，从中收集所需的信息并产生HTML页面。它们还可以定期执行这样的动作，从而不断更新页面。（可以到CNN.com和Google News上面看看“last generated”（最近产生的）信息。）

为什么要使用数据库而不使用简单的文本文件呢？原因有4点：

- 数据库速度更快。数据库会把跟踪关键信息（如姓氏或ID号）的索引（index）保存在文件中，因此，你可以快速查到“Guzdial”。文件通过文件名来索引，而非通过文件中的内容。
- 数据库是标准化的。可以用多种工具或语言访问Microsoft Access、Informix、Oracle、Sybase和MySQL数据库。
- 数据库可以是分布式的。位于不同计算机上的大量用户可以通过网络把信息存入数据库或者从数据库中取出信息。
- 数据库存储关系（relations）。使用列表表示像素时，我们必须在脑子里记住每个数字分别是什么意思。数据库可以保存数据字段（field）的名称。如果数据库知道哪些字段比较重要（比如你最有可能基于哪些字段来搜索），那么它就可以在这些字段上建立索引。

Python对多种数据库提供了内置支持，它还提供了一种称为anydbm的可应用于任意数据库的通用支持（如图12.8所示）。键（key）可以表示在方括号中，访问这些字段时速度最快。如果用anydbm，那么键和值（value）都只能是字符串。下面是一个从anydbm中取出信息的例子。

```
>>> db = anydbm.open("mydbm","r")
>>> print db.keys()
['barney', 'fred']
>>> print db['barney']
My wife is Betty.
>>> for k in db.keys():
...     print db[k]
...
My wife is Betty.
My wife is Wilma.
>>> db.close()
```

```
>>> import anydbm
>>> db = anydbm.open("mydbm","c")
>>> db["fred"] = "My wife is Wilma."
>>> db["barney"] = "My wife is Betty."
>>> db.close()
```

anydbm是Python
内置的数据库

“创建”数据库

数据库在这些键
上建立索引

图12.8 使用简单的数据库

Python中的另外一种数据库，shelve，允许我们将字符串、列表、数字甚至其他任何东西作为值来保存。

```
>>> import shelve
>>> db=shelve.open("mysshelf","c")
>>> db["one"]=["This is",["a","list"]]
>>> db["two"]=12
>>> db.close()
>>> db=shelve.open("mysshelf","r")
>>> print db.keys()
['two', 'one']
>>> print db['one']
['This is', ['a', 'list']]
>>> print db['two']
12
```


12.3.1 关系型数据库

现代数据库大多是关系型数据库 (relational database)。关系型数据库使用表 (table) 来存储信息 (如图12.9所示)。在关系式的表中, 列有名字, 且认为一行行的数据是彼此关联的信息。

复杂的关系要用多张表来保存。假定有一些学生的照片, 想管理哪些学生出现在哪些照片上的信息——一张照片上可能有多个学生。你可以用多张表来表示这样一种结构: 一张表记录学生和学生的ID, 一张表记录照片和照片ID, 另一张表保存学生ID和照片ID之间的映射关系, 如图12.10所示。

Name	Age
Mark	40
Matthew	11
Brian	38

图12.9 关系表示例

Picture	PictureID
Class1.jpg	P1
Class2.jpg	P2

StudentName	StudentID
Katie	S1
Brittany	S2
Carrie	S3

PictureID	StudentID
P1	S1
P1	S2
P2	S3

图12.10 用多张表表示更加复杂的关系

如何用图12.10中的表来得出Brittany出现在哪些照片中呢? 可以先在学生表中查询Brittany的ID, 然后在“照片-学生”表中查找照片ID, 最后在照片表中查找照片的名字, 得到结果: Class1.jpg。如果要得出这张照片中有哪些人, 又该怎么做呢? 可以首先在照片表中查出ID, 然后查出哪些学生的ID与这张照片的ID关联, 最后查出学生的名字。

这种使用多张表来回答一个查询 (query) (从数据库中取得信息的请求) 的做法称为联接 (join)。在数据库表保持简单, 每行只代表一种关系的情况下, 数据库联接的效果最佳。

12.3.2 基于散列表的关系型数据库示例

为解释关系型数据库的思想, 本节将用Python中的一种更简单的结构来构造一个关系型数据库。通过这种方法我们可以描述某些数据库思想 (比如联接) 的工作原理。本节属于选学内容。

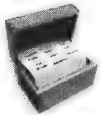
我们可以用一种称为散列表 (hash table) 或字典 (dictionary) 的结构 (其他一些语言也称之为关联式数组), 通过shelve来创建数据库关系表中的行。与数据库类似, 散列表支持键和值的关联, 但数据只存在于内存中。

```
>>> row={'StudentName':'Katie','StudentID':'S1'}
>>> print row
{'StudentID': 'S1', 'StudentName': 'Katie'}
>>> print row['StudentID']
S1
>>> print row['StudentName']
Katie
```

除了一次性定义完整的散列表外，我们还可以一点点地填充它。

```
>>> pictureRow = {}
>>> pictureRow['Picture']='Class1.jpg'
>>> pictureRow['PictureID']='P1'
>>> print pictureRow
{'Picture': 'Class1.jpg', 'PictureID': 'P1'}
>>> print pictureRow['Picture']
Class1.jpg
```

现在，我们可以创建一个关系型的数据库了，方法是将每张表存储在一个单独的shelve数据库中，用散列表表示每一行。



程序113：使用shelve创建关系型数据库

```
import shelve
def createDatabases():
    # 创建学生数据库
    students=shelve.open("students.db","c")
    row = {'StudentName':'Katie','StudentID':'S1'}
    students['S1']=row
    row = {'StudentName':'Brittany','StudentID':'S2'}
    students['S2']=row
    row = {'StudentName':'Carrie','StudentID':'S3'}
    students['S3']=row
    students.close()
    # 创建照片数据库
    pictures=shelve.open("pictures.db","c")
    row = {'Picture':'Class1.jpg','PictureID':'P1'}
    pictures['P1']=row
    row = {'Picture':'Class2.jpg','PictureID':'P2'}
    pictures['P2']=row
    pictures.close()
    # 创建“照片~学生”数据库
    pictures=shelve.open("pict-students.db","c")
    row = {'PictureID':'P1','StudentID':'S1'}
    pictures['P1S1']=row
    row = {'PictureID':'P1','StudentID':'S2'}
    pictures['P1S2']=row
    row = {'PictureID':'P2','StudentID':'S3'}
    pictures['P2S3']=row
    pictures.close()
```

程序原理

createDatabases函数实际上创建了三张不同的数据库表。

- 第一张表是students.db。创建了表示Katie的字典（散列表），Katie的ID是“S1”，然后把这个字典对象保存在数据库students中，用学生的ID“S1”作为索引。然后对Brittany和Carrie也做同样处理——完全可以使用同一个row变量表示它们，因为我们只是创建了

一张散列表，然后就把它存到数据库里去了。

- 接下来创建了pictures.db数据库。建立了“照片”Class1.jpg和它的ID“P1”之间的关系，并将这一关系存进pictures数据库中。又对照片Class2.jpg重复这一过程。可否将同一个变量database用于各个不同的数据库呢？当然可以，因为我们每次只打开一个数据库，打开下一个之前即关闭它。但那样一来程序的可读性会差一些。
- 最后，我们打开并填充了pict-students.db数据库。创建了关联照片ID和学生ID的行，将它们保存在数据库中，然后关闭了数据库。

创建这样一个数据库后，我们就可以做一次联接（join）查询了。显然，这里的思想是我们在建好数据库之后有时会执行这种查询，或许在查询之前又追加了大量数据项。（如果数据库中只有两张照片和三个学生，那它岂不成了一道很傻的编程习题？）必须遍历数据库中的数据来找出匹配查询需求的值。



程序114：使用shelve数据库执行联接查询

```
def whoInClass1():
    # 获得照片ID
    pictures=shelve.open("pictures.db","r")
    for key in pictures.keys():
        row = pictures[key]
        if row['Picture'] == 'Class1.jpg':
            id = row['PictureID']
    pictures.close()
    # 获得学生的ID
    studentslist=[]
    pictures=shelve.open("pict-students.db","c")
    for key in pictures.keys():
        row = pictures[key]
        if row['PictureID']==id:
            studentslist.append(row['StudentID'])
    pictures.close()
    print "We're looking for:",studentslist
    # 获得学生的名字
    students = shelve.open("students.db","r")
    for key in students.keys():
        row = students[key]
        if row['StudentID'] in studentslist:
            print row['StudentName'], "is in the picture"
    students.close()
```

程序原理

联接查询的每一部分都需要对数据库做不同的循环。

- 首先，打开了pictures.db并将它命名为pictures。我们循环遍历所有的键，获得相应的每一行（散列表），然后检查其中的'Picture'字段是否为Class1.jpg。如果找到，便把照片的ID保存在变量id中。用完数据库即关闭它。
- 然后，用变量pictures打开了pict-students.db（重用这个变量名没有问题）。我们知道，一张照片中可能有多个学生，因此，我们创建了一个列表studentslist来保存找到的学生ID。对于pictures中的各个键，我们从散列表中查看'PictureID'一项，看它是否与我们的id变量匹配。如果能找到与之匹配的变量，那么就把散列表中的'StudentsID'附加到studentslist列表中。

- 最后，我们用变量`students`打开了`students.db`数据库。遍历所有的键并取得表示每一行数据的散列表`row`。我们使用了列表的另一项漂亮特性：直接询问一个字符串是否存在于（in）`studentslist`列表中，这种用法我们之前从未见过。如果给定的'`StudentID`'是我们要找的ID之一（存在于`studentslist`中），那么就将哈希表中相应的'`StudentName`'项打印出来。最后，我们关闭了`students`数据库以完成清理工作。

上面的函数可以这样运行：

```
>>> whoInClass1()
We're looking for: ['S2', 'S1']
Brittany is in the picture
Katie is in the picture
```

12.3.3 使用SQL

真正的数据库不会让你用循环来做联接查询。相反，通常可以用结构化查询语言（Structured Query Language, SQL）来处理并查询数据库。实际上，SQL数据库语言家族中有好几种语言，但我们不想在这里区分它们。SQL是一种庞大而又复杂的编程语言，这里根本就没打算全面介绍它。但我们还是会讲到足够多的信息，让你对SQL的结构和用法有个基本的了解。

SQL可用于多种数据库，包括Microsoft Access。使用我们即将介绍的方法，Python几乎可以与任何数据库系统打交道。另外还有一些可以免费获得的数据库，比如MySQL，它们也使用SQL，而且可以用Python来控制。如果想摆弄一下这里的例子，你可以把MySQL安装到JES中，MySQL可以从<http://www.mysql.com>获取。你需要配置MySQL，而且要下载支持Java访问MySQL的JAR文件并将它放到Jython Lib文件夹下。

要从JES中操作MySQL数据库，你需要创建一个连接（connection），它将为你提供一个可以用SQL来操作的游标（cursor）。在JES中使用MySQL的时候，会使用一个`getConnection`函数，以便执行`con = getConnection()`这样的语句。



程序115：从Jython中获取一个MySQL连接

我们觉得所有这些细节都太难记了，因此将它们全部隐藏在一个函数中，调用时只需写`con = getConnection()`，但程序区要有以下内容：

```
com.ziclix.python.sql
import zxJDBC
def getConnection():
    db = zxJDBC.connect("jdbc:mysql://localhost/test", "root", None, "com.mysql.jdbc.Driver")
    con = db.cursor()
    return con
```

从这里开始，要执行SQL命令，可以从连接对象上调用`execute`方法，就像这样：

```
con.execute("create table Person (name VARCHAR(50), age INT)")
```

以下是SQL的简单介绍：

- 要创建表格，使用SQL命令：`create table tablename(columnname datatype, ...)`。所以，在上面的例子中，我们将创建一个`Person`表，它有两列：一列是名字（name），它由数量可变的字符组成，最多50个字符；另一列是用整数表示的年龄（age）。其他的

数据类型还有：numeric、float、date、time、year和text。

- 要往数据库中插入数据，使用命令：insert into tablename values (columnvalue1, columnvalue2, ...)。在下面的例子中，使用多种引号方式显得非常方便。

```
con.execute('insert into Person values
("Mark",40)')
```

- 从数据库中选择数据可以直接理解为从数据库表中选择自己想要的行，就像从字处理器或电子表格软件中选择数据一样。下面是一些使用SQL选择数据的例子：

```
Select * from Person
Select name,age from Person
Select * from Person where age>40
Select name,age from Person where age>40
```

我们可以用Python完成所有这些任务。连接对象上有一个**实例变量**（instance variable）（与方法差不多，只有这个类的对象知道它）：rowcount，它能告诉你选出的行数。针对选出的数据，fetchone()方法将以元组（tuple）（可以理解为一种特殊的列表——多数情况下我们就把它当列表来用）的形式返回下一行。

```
>>> con.execute("select name,age from Person")
>>> print con.rowcount
3
>>> print con.fetchone()
('Mark', 40)
>>> print con.fetchone()
('Barb', 41)
>>> print con.fetchone()
('Brian', 36)
```

我们还可以使用条件选择。



程序116：使用条件选择选取并显示数据

```
def showSomePersons(con, condition):
    con.execute("select name, age from Person "+condition)
    for i in range(0,con.rowcount):
        results=con.fetchone()
        print results[0]+" is "+str(results[1])+" years old"
```

程序原理

showSomePersons函数接受一条数据库连接con和一个条件作为输入，其中的条件必须是包含SQL子句的字符串。然后让连接执行（execute）SQL select命令。执行时，我们把condition连接到select命令的末尾。然后用一个循环来遍历选择操作返回的每一行（使用con.rowcount获得行数）。我们用fetchone取得各行并打印出结果。注意我们在返回的元组上使用的是常规的索引列表的方法。

以下是使用showSomePersons的一个例子：

```
>>> showSomePersons(con,"where age >= 40")
Mark is 40 years old
Barb is 41 years old
```

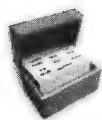
现在，我们可以考虑使用条件选择来实现一种连接查询。以下代码片段的含义是：从这三张表中返回照片和学生名字，其中学生的名字是“Brittany”，学生的ID要与“学生-照片”ID映射表中的学生ID相同，并且ID映射表中的照片ID要与图片表中的ID相同。

```
Select
    p.picture,
    s.studentName
From
    Students as s,
    IDs as i,
    Pictures as p
Where
    (s.studentName="Brittany") and
    (s.studentID=i.studentID) and
    (i.pictureID=p.pictureID)
```

12.3.4 使用数据库构建Web页面

现在，让我们再回到先前的网页产生器程序。我们可以把信息存放在数据库中，需要时把它们从数据库中取出并写到Web页面中去——Amazon、CNN和eBay也是这么做的。

```
>>> import anydbm
>>> db=anydbm.open("news","c")
>>> db["headline"]="Katie turns 8!"
>>> db["story"]="Our daughter, Katie, turned 8 years
old yesterday. She had a great birthday. Grandma and
Grandpa came over. The previous weekend, she had
three of her friends over for a sleepover then a
morning run to Dave and Buster's."
>>> db.close()
```



程序117：从数据库中获取信息来构造Web页面

```
def makeHomePage(name, interest):
    file=open("homepage.html","wt")
    file.write(doctype())
    file.write(title(name+"'s Home Page"))
    # 导入数据库内容
    db=anydbm.open("news","r")
    file.write(body("""
<h1>Welcome to """+name+"""'s Home Page</h1> <p>Hi! I am
"""+name+""". This is my home page! I am interested in
"""+interest+""")
    </p> <p>Right here and right now it's
"""+findTemperatureLive()+"" degrees. (If you're in the
North, nyah-nyah!).</p>
<p>Random thought for the day:
"""+sentence()+""</p>
<h2>Latest news:
"""+db["headline"]+""</h2>
<p>"""+db["story"]+""</p>""))
    file.close()
    db.close()

# 剩下的部分，比如findTemperatureLive(), 与之前的例子相同
```

现在我们可以想一想，像CNN这样的大型网站是如何工作的。记者将新闻报道录入到分布于世界各地的数据库中。编辑（可能分布在不同地方，也可能集中在一个地方）从数据库中取出报道，校正一下再存回去。一个产生Web页面的程序定期运行（出现热点新闻时可能运行得频繁一些），收集数据库中的新闻报道并产生HTML。“哇！你已经有一个大型网站了！”数据库对于大型网站的运行至关重要。

习题

- 12.1 SGML、HTML、XML和XHTML之间的区别是什么？
- 12.2 将下列十六进制数转换成十进制：2A3、321、16、24、F3。
- 12.3 将下列十进制数转换成十六进制：113、64、129、72、3。
- 12.4 将下列颜色值转换成十六进制表示：gray、yellow、pink、orange和magenta。你可以用 `print color` 获得各种颜色的红、绿、蓝分量。
- 12.5 编写函数创建一张简单的主页。上面要有你的名字、照片，以及一张包含选修课程和任课教师的表格。
- 12.6 编写函数从 `http://www.cnn.com` 上读取当前的头条新闻并返回。修改创建主页的函数，把这个函数用上去。
- 12.7 编写函数从朋友的Facebook页面中读取当前状态并返回。修改创建主页的函数，使用这个函数来显示5个朋友在Facebook主页上的状态信息。
- 12.8 使用散列表来保存一些快捷文字及其定义。比如“lol”表示“laugh out loud”。使用这张散列表解码一条文本消息。
- 12.9 假如有一个关系型数据库，其中有一张人员表，它有id、name和age等字段。以下各条语句分别返回什么结果？
 - `Select * from person`
 - `Select age from person`
 - `Select id from person`
 - `Select name, age from person`
 - `Select * from person where age > 20`
 - `Select name from person where age < 20`
- 12.10 为什么我们应该使用关系型数据库？还有其他类型的数据库吗？关系型数据库是谁发明的？
- 12.11 关系型数据库问题：
 - 什么是数据库表？
 - 什么是联接（join）？
 - 什么是查询？
 - 什么是连接（connection）？
- 12.12 关系型数据库问题：
 - 什么是SQL？
 - 如何用SQL创建表格？
 - 如何用SQL在表格中插入一行？
 - 如何用SQL从表格中取得数据？

- 12.13 你爸爸打电话给你：“我们的技术支持说公司的网站挂了，原因是数据库程序出了故障。数据库与公司网站有什么关系？”跟他解释一下为什么数据库是运行大型网站的必备部分。要解释两点：(a) 网站是如何基于数据库来构建的；(b) HTML是如何实际生成的。
- 12.14 你有了一台新计算机，看起来可以连到因特网上，然而，当你试着访问 `http://www.cnn.com` 时，却只得到一个“Server Not Found”错误信息。你给技术支持打电话，他们告诉你试试 `http://64.236.24.20`。这次好了。这时你和技术支持都明白了是计算机的配置出了问题。你可以通过因特网访问网站，而域名 `www.cnn.com` 却无法得到识别，这是什么问题呢？
- 12.15 为某个图片文件夹创建索引HTML页面，页面上要有指向各个图片的链接。编写一个函数，接受一个字符串参数，参数即为该目录的路径。你需要在文件夹下创建一个名为 `index.html` 的HTML页面，页面中包含指向目录中各幅图片的链接。你还应该为各幅图片产生缩略版本（大小为原图的一半）。可以用 `makeEmptyPicture` 创建适当大小的空白图片，然后将原图缩小存到其中。新图片的名字可以用“half-”加上原文件的名字（比如原文件名为“fred.jpg”，尺寸减半的图片可以另存为“half-fred.jpg”）。在锚中使用尺寸减半的图片作为指向原始图片的链接。
- 12.16 为HTML主页产生器增加一项功能：根据气温产生随机评论。
- 如果气温低于32°F (0°C——译者注)，可以插入“当心冰冻！”或者“会下雪吗？”
 - 如果气温在32°F和50°F (10°C——译者注) 之间，可以插入“冬天快过去吧，我等不及了！”或者“春天快来吧！”
 - 如果气温在50°F到80°F (约27°C——译者注) 之间，可以插入“天气转暖了！”或者“适合穿单外套的天气。”
 - 如果气温超过80°F，可以插入“夏天，终于来了！”或者“是游泳的季节了！”
- 写一个函数 `weathercomment`，接受气温作为输入，随机返回上述评论。换种说法：你应该根据气温决定哪些句子是恰当的，然后从中随机选取一个。
- 12.17 编写一个函数，从包含名字和电话号码的文件中读取已分隔开的字符串，然后使用数据库将名字作为键，电话号码作为值保存起来。接受文件名和人名作为输入，查找这个人的电话号码。如果已知电话号码要查找它所属的人名，又该怎么做呢？
- 12.18 创建一个关系型数据库，其中包含人物表 (`person`)、照片表 (`picture`) 和人物-照片表 (`person-picture`)。人物表中存有ID、姓名和年龄。照片表中存有照片ID和文件名。人物-照片表中记录着每张照片上的人。编写一个函数，确定哪些照片上有超出某年龄的人。
- 12.19 创建一个关系型数据库，其中包含产品表 (`product`)、客户表 (`customer`)、订单表 (`order`) 和订单项目表 (`order item`) 各一张。产品表中存有ID、名称、图片、描述和价格。客户表中存有ID、姓名和地址。订单表中存有订单ID和客户ID。订单项目表中存有订单ID、产品ID和数量。编写一个函数找出特定客户的所有订单。再编写一个函数找出总价大于某特定值的所有订单。

深入学习

关于使用Python或Jython进行Web开发和编程的好书有很多。我们特别推荐Gupta[18]和Hightower[24]的书，因为它们数据库的情景中讨论了Java的使用。

电 影

第13章 制作和修改电影

制作和修改电影

本章学习目标

本章媒体学习目标：

- 理解一系列静止的图像是如何被感知成动作的。
- 以不同的运动方式和效果制作动画。
- 在动画处理中使用视频源。
- 弄清数字效果的实现原理。

本章计算机科学学习目标：

- 理解使用多个函数简化编程的另一个例子。
- 理解自底向上设计及实现的一个详细例子。
- 使用Python函数的可选参数和命名参数。

说实话，电影（视频）很容易处理。它们是图片（帧）组成的数组。你需要知道帧频（frame rate，每秒钟播放的帧数），但它与你以前见过的某个概念非常类似。我们将使用电影（movie）这一术语来泛指动画（animation，完全由数字绘图产生的动作）和视频（video，由某种摄像过程产生的动作）。

使电影成为可能是人类视觉系统的一个称为视觉暂留（persistence of vision）的特性。我们并不能看清世界上发生的一切变化。比如，你通常看不到自己在眨眼，虽然它们是如此频繁地发生着（通常一分钟20次）。我们不会在每次眨眼时都惊慌失措地想：“世界哪儿去了？”相反，我们的眼睛在一小段时间里保持了一副图像并不断告诉大脑原来的那幅图像还在。

如果我们以足够快的速度看到两幅相关的图像，眼睛就会保留图像，而大脑就看到了持续的运动。“足够快”的定义是每秒钟大约16帧——我们在一秒内看到16幅相关的图片，就会以为它是连续的运动。如果图片内容不相关，大脑便会报告一次蒙太奇（montage），即一组彼此无关的图像（虽然可能在主题上有联系）。我们将这个“每秒16帧”（16 fps）称为运动感觉的下限。

早期的无声动画就是16 fps的。后来，为了让声音更平滑，动画标准规定了24 fps——在16 fps的速度下胶卷上无法提供足够的物理空间来编码声音数据。（有没有疑惑过无声动画为什么看起来更快或更不平稳？考虑一下缩放图片或声音时的效果——当以24 fps的速度播放16 fps的电影时，情形完全一样。）数字视频设备（比如数码摄像机）以30 fps的速度捕捉画面。高速拍摄又有何用呢？美国空军的某些实验表明，飞行员可以在1/200秒的瞬间识别出以飞机形状呈现的短暂亮光（并判断出它是哪种机型）。视频游戏的玩家们则说他们能觉出30 fps视频和60 fps视频之间的差别。

由于数据量大、速度快，电影的处理颇具挑战性。视频的实时处理（比如，随着视频的播

放，在每一帧画面进入或退出时做相应处理）更不容易，因为任何处理动作都必须在1/30秒的时间内完成。我们来计算一下录制视频需要的字节数量：

- 假如一帧画面的尺寸为 640×480 像素，速度为30 fps，则一秒钟的图像需要 30 （帧） $\times 640 \times 480$ （像素） $= 9\,216\,000$ 像素。
- 使用24位彩色（R、G、B各占一字节），那就是 $27\,648\,000$ 字节，或者说27 MB。
- 对一部90分钟的故事片来说，那就是 $90 \times 60 \times 27\,648\,000 = 149\,299\,200\,000$ 字节——149 GB。

数字电影差不多都是以压缩格式存储的。一张DVD只能存储6.47 GB，因此即使在DVD上，电影也是压缩的。像MPEG、QuickTime和AVI这类电影格式标准都使用压缩格式。它们并不记录每一帧画面——而是记录一些关键帧，然后记录前后两帧的差异。JMV格式稍有不同——它是JPEG图像组成的文件，因此每一帧都存在于文件中，且每一帧都是压缩的。

与图片和声音相比，电影可以用一些不同的压缩技术。考虑某个人走路的画面，连续的两帧画面之间，变化只有一点点——人物先前的位置和后来的位置不一样。如果我们只记录这些差异，而不是整帧画面的像素，就可以节省大量空间。

MPEG电影实际上就是一列MPEG图像跟一个音频文件（比如MP3）的合并。我们将沿着这条线索来介绍电影，而且不处理音频。下一节介绍的工具能创建带声音的电影，然而，在电影的处理中，真正麻烦的部分还是对所有图像的处理。我们关注的也是这个。

13.1 产生动画

为制作电影，我们将创建一系列JPEG帧，然后把它们组装起来。我们将把所有的帧放置在一个目录下并为它们编号，这样，工具程序就能知道如何以正确的次序将它们组装成电影。我们将把文件依次命名为frame01.jpg，frame02.jpg……由于帧数较多，在数字开头补足一个0很重要，这样，当文件以字母表顺序排列时，实际上也是以数字顺序排列的。

下面是第一个产生电影的程序，它只是让一个红色方块从左上角移至右下角（如图13.1所示）。

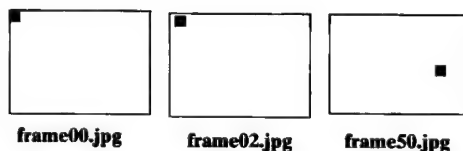
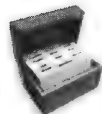


图13.1 从第一段电影中取出的几帧：将一个方块往右下方移动



程序118：制作移动方块的电影

```
def makeRectMovie(directory):
    for num in range(1,30): # 29帧 (1到29)
        canvas = makeEmptyPicture(300,200)
        addRectFilled(canvas,num * 10, num * 5, 50,50, red)
        # 数字转换成字符串
        numStr=str(num)
        if num < 10:
            writePictureTo(canvas,directory+"\\frame0"+numStr+".jpg")
        if num >= 10:
            writePictureTo(canvas,directory+"\\frame"+numStr+".jpg")
```

```
movie = makeMovieFromInitialFile(directory+"\\frame00.jpg");
return movie
```

你可以尝试一下这个程序，比如在Temp目录下创建一个rect目录，然后执行下面的命令：

```
>>> rectM = makeRectMovie("c:\\Temp\\rect")
>>> playMovie(rectM)
```

执行playMovie(movie)命令时，电影的所有画面将在一个电影播放器程序中播放。播放结束时，你可以用Prev按钮查看前一帧，或使用Next按钮查看后一帧。按Play Movie按钮可以再次播放整段电影。Delete All Previous按钮将删除目录中当前帧之前的所有帧。Delete All After按钮将删除目录中当前帧之后的所有帧。Write Quicktime按钮将基于目录中的各帧图像输出一部QuickTime电影。Write Avi按钮则基于目录中的各帧图像输出一部AVI电影。产生的电影存放于帧图像所在的目录，名字与目录相同。

程序原理

这一菜谱的关键部分是makeEmptyPicture命令之后的那几行代码。必须根据当前帧号计算方块的不同位置。调用addRectFilled()函数时，使用的公式根据电影中的不同帧计算不同的位置（如图13.2所示）。

如果你尝试用setPixel()设置图片边界之外的像素，setPixel会很不高兴。然而，像addText()和addRect()这样的函数却不会因为结果超出图像边界而产生错误。它们只会基于图片的尺寸来裁剪图像（只显示能够显示的部分），因此，你可以编写简单的代码来构造动画，不必担心越界问题。这着实简化了纸带电影的制作（如图13.3所示）。

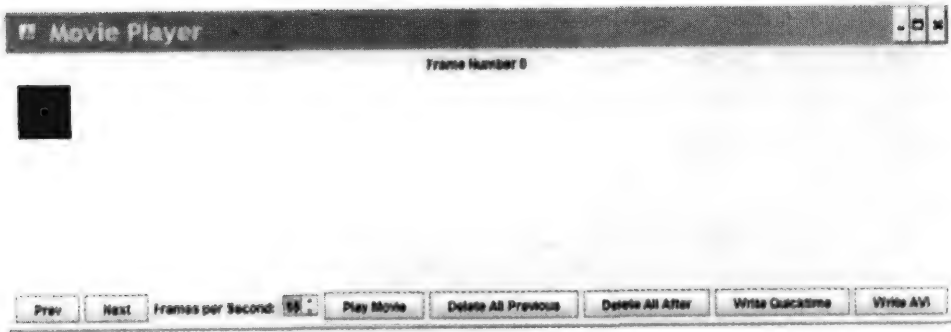


图13.2 播放方块电影的电影播放器

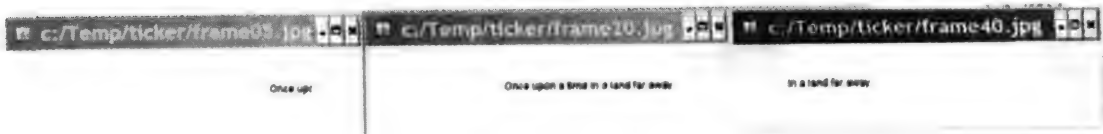


图13.3 纸带电影



程序119：产生纸带电影

```
def tickertape(directory,string):
    for num in range(1,100): # 99帧
        canvas = makeEmptyPicture(300,100)
        # 从右侧开始向左移动
        addText(canvas,300-(num*10),50,string)
        # 现在输出一帧画面
        # 必须正确处理一位数字和两位数字的不同情况
        differently
        numStr=str(num)
        if num < 10:
            writePictureTo(canvas,directory+"//frame0"+numStr+".jpg")
```

~这几行程序应该与下面的一行连续。Python中一条命令不可以跨多行。

```
if num >= 10:
    writePictureTo(canvas,directory+"//frame"+numStr+".jpg")
```

程序原理

tickertape接受一个目录和一个输出到电影中的字符串参数，电影中的每一帧图像都存在第一个参数指定的目录中。针对99帧画面（1~100但不包括100），我们分别构造一幅300×100的空白图片，然后将参数字符串作为文本绘制到图片上。字符串的y坐标都是50（垂直位置相同），而x坐标（水平位置）等于300 - (num×10)。随着帧号num的递增，这一表达式的值将递减。因此，越后面的帧字符串的位置越靠近左边（更小的x值）。我们把帧号num转换成字符串numStr并使用这个字符串构造一个文件名，其中帧号部分需要时以0补足。

可不可以同时移动多个物体呢？当然可以！只是绘图代码会复杂一些。我们还是可以像先前那样，让物体做线性移动，但这一次我们将尝试一种不同的方式。下面的菜谱使用正弦和余弦函数创建圆形的运动轨迹，同时伴随着程序118中的线性移动（如图13.4所示）。



程序120：同时移动两个物体

```
def movingRectangle2(directory):
    for num in range(1,30): # 29帧
        canvas = makeEmptyPicture(300,250)
        # 添加一个做线性移动的实心方块
        addRectFilled(canvas,num*10,num*5, 50,50,red)

        # 添加一个转动的方块
        blueX = 100+ int(10 * sin(num))
        blueY = 4*num+int(10* cos(num))
        addRectFilled(canvas,blueX,blueY,50,50,blue)

        # 现在输出一帧画面
        # 必须正确处理一位数字和两位数字的不同情况
        numStr=str(num)

        if num < 10:
            writePictureTo(canvas,directory+"//frame0"+numStr+".jpg")
        if num >= 10:
            writePictureTo(canvas,directory+"//frame"+numStr+".jpg")
```

程序原理

我们知道，sin和cos产生-1~1之间的值。（你还记得，对吗？）蓝色方块的x坐标置为

$100 + \text{int}(10 * \sin(\text{num}))$)。这就是说蓝色方块的 x 坐标将在100附近, 加减不超过10个像素。随着 \sin 值的变化, 它将从左向右, 然后又从右向左移动。 y 坐标的位置则由公式 $4 * \text{num} + \text{int}(10 * \cos(\text{num}))$ 确定。这样, y 坐标将一直递增(方块在下降), 但下降的幅度也会增减不超过10个像素, 于是它的下落过程会伴随着轻度的上下移动。

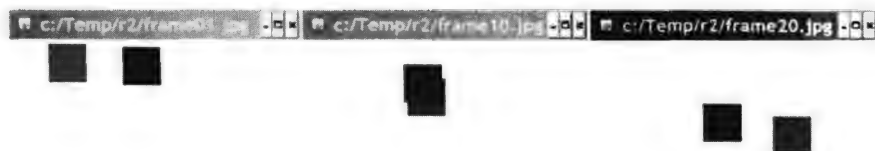


图13.4 同时移动两个方块

制作动画时, 我们不仅可以使⽤基础的绘图函数, 还可以通过`setColor()`来使⽤以前用过的那种图片, 只是那样的代码运行起来会很慢。



调试技巧: 使用`printNow`产生即时输出

JES有一个`printNow()`函数, 它接受一个字符串并把它立即打印出来——而不会等到函数完成才把一行内容打印到命令区。如果你想知道函数现在处理到第几帧了, 那么这一点很有用。你会考虑在画面出来时, 双击一下, 从操作系统中看一看前面的几帧。

下面的菜谱让Mark的头像在屏幕上四处游荡。这个函数在计算机上需执行一分钟才能结束。图13.5显示了程序运行时输出画面中的两帧。

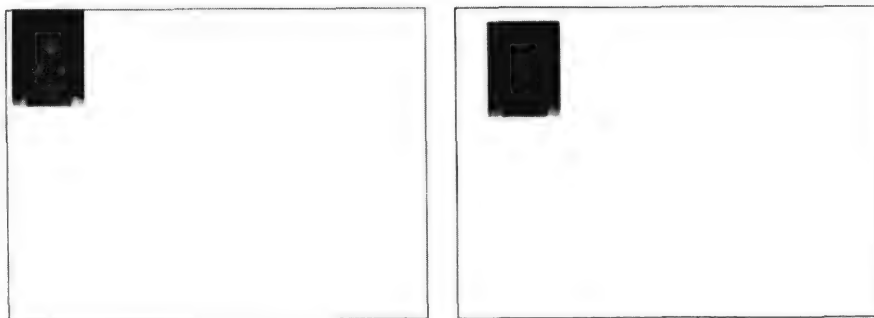


图13.5 移动头像电影中的两帧



程序121: 移动Mark的头像

```
def moveHead(directory):
    markF=getMediaPath("blue-mark.jpg")
    mark = makePicture(markF)
    head = clip(mark,275,160,385,306)
    for num in range(1,30): # 29帧
        printNow("Frame number: "+str(num))
        canvas = makeEmptyPicture(640,480)
        # 现在执行实际复制
        copy(head,canvas,num*10,num*5)
        # 现在输出一帧画面
```

```

# 必须正确处理一位数字和两位数字的不同情况
numbers_differently
numStr=str(num)
if num < 10:
    writePictureTo(canvas,directory+"//frame0"+numStr+".jpg")
if num >= 10:
    writePictureTo(canvas,directory+"//frame"+numStr+".jpg")

def clip(picture,startX,startY,endX,endY):
    width = endX - startX + 1
    height = endY - startY + 1
    resPict = makeEmptyPicture(width,height)
    resX = 0
    for x in range(startX,endX):
        resY=0 # 重新设定结果的y下标
        for y in range(startY,endY):
            origPixel = getPixel(picture,x,y)
            resPixel = getPixel(resPict,resX,resY)
            setColor(resPixel,(getColor(origPixel)))
            resY=resY + 1
        resX=resX + 1
    return resPict

```

程序原理

我们创建了一种新的clip方法，它构造并返回了一幅新图片，其中只包含由输入参数startX、startY、endX和endY定义的矩形范围内的像素。我们用这个clip方法创建一幅只包含Mark头像的图片。然后，我们使用通用的copy函数（程序30），根据帧号num把Mark的头像复制到画布canvas的不同位置。

到目前，电影程序越来越复杂了。我们希望把程序分开写到不同的部分，并且让输入帧画面的部分保持独立。这意味着我们可以在函数主体中只关注想要的画面内容，而不用关心怎样输出。这是过程式抽象的例子。



程序122：移动Mark的头像：简化版本

```

def moveHead2(directory):
    markF=getMediaPath("blue-mark.jpg")
    mark = makePicture(markF)
    face = clip(mark,275,160,385,306)
    for num in range(1,30): # 29帧
        printNow("Frame number: "+str(num))
        canvas = makeEmptyPicture(640,480)
        # 现在执行实际复制
        copy(face,canvas,num*10,num*5)
        # 现在输出一帧画面
        writeFrame(num,directory,canvas)

def writeFrame(num,dir,pict):
    # 必须正确处理一位数字和两位数字的不同情况
    numStr=str(num)
    if num < 10:
        writePictureTo(pict,dir+"//frame0"+numStr+".jpg")
    if num >= 10:
        writePictureTo(pict,dir+"//frame"+numStr+".jpg")

```

但是，这个writeFrame()函数假定使用最多两位数字的帧号。我们想要的帧数可能更多，下面的版本支持三位数字的帧号。



程序123: writeFrame()100帧以上

```
def writeFrame(num,dir,pict):
    # 必须正确处理一位数字、两位数字和三位数字的不同情况
    numStr=str(num)
    if num < 10:
        writePictureTo(pict,dir+"//frame00"+numStr+".jpg")
    if num >= 10 and num<100:
        writePictureTo(pict,dir+"//frame0"+numStr+".jpg")
    if num >= 100:
        writePictureTo(pict,dir+"//frame"+numStr+".jpg")
```

我们可以把第3章的图像处理程序用于多帧画面，从而做出相当有趣的电影。还记得产生日落效果的程序吗（程序12）？现在，我们把它修改一下，最终用多帧画面产生缓缓の日落效果。我们把程序改成让前后两帧之间只有1%的差异。实际上，这一版程序做过头了，它产生了一颗“超新星”，但效果还是很有趣的（如图13.6所示）。

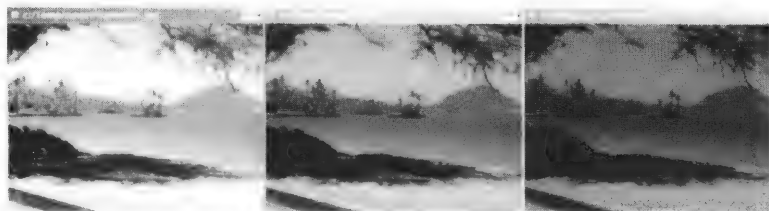


图13-6 缓缓日落电影的多帧画面



程序124: 制作缓缓日落的电影

```
def slowSunset(directory):
    # 循环之外!
    canvas = makePicture(getMediaPath("beach-smaller.jpg"))
    for num in range(1,100): # 99帧
        printNow("Frame number: "+str(num))
        makeSunset(canvas)
        # 现在输出一帧画面
        writeFrame(num,directory,canvas)

def makeSunset(picture):
    for p in getPixels(picture):
        value=getBlue(p)
        setBlue(p,value*0.99) # 只减少1%
        value=getGreen(p)
        setGreen(p,value*0.99)
```

程序原理

制作这部电影的关键之处在于：在创建各帧画面的循环开始之前先创建画布canvas。在循环内部，反复使用这块画布。于是每次调用makeSunset时，都把“日落度”（有这么个词么？）增加了1%。（我们打算再列出writeFrame()了，我们假定你会在程序区和程序文件中包含它。）

以前实现的swapBack菜谱也可以用在电影的制作中并产生不错的效果。我们可以修改程序45，接受一个阈值作为输入，调用时我们把帧号作为阈值传入。效果就是画面中前景缓缓淡出，最后只剩下背景图像（如图13.7所示）。



程序125: 缓缓淡出

```
def swapBack(pic1, back, newBg, threshold):
    for x in range(0,getWidth(pic1)):
        for y in range(0,getHeight(pic1)):
            p1Pixel = getPixel(pic1,x,y)
            backPixel = getPixel(back,x,y)
            if (distance(getColor(p1Pixel),getColor(backPixel))
                < threshold):
                setColor(p1Pixel,getColor(getPixel(newBg,x,y)))
    return pic1

def slowFadeout(directory):
    origBack = makePicture(getMediaPath("bgframe.jpg"))
    newBack = makePicture(getMediaPath("beach.jpg"))
    for num in range(1,60): #59 frames
        # 在循环中构造小孩图片
        kid = makePicture(getMediaPath("kid-in-frame.jpg"))
        swapBack(kid,origBack,newBack,num)
        # 现在输出一帧画面
        writeFrame(num,directory,kid)
```

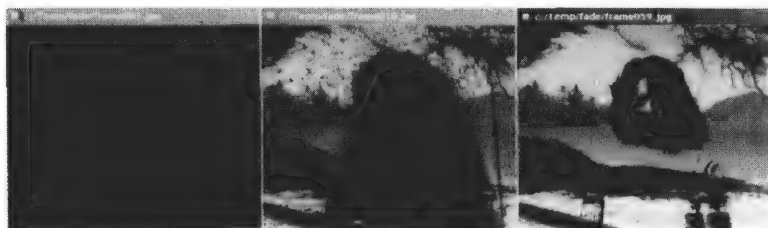


图13.7 “缓缓淡出”电影中的几帧画面

程序原理

在这个程序中，帧号是阈值，它决定我们在swapBack中何时换入新的背景而不再保留原像素。随着阈值的递增，越来越多地用新背景中的像素替换原图中的像素。于是，随着帧号的增加，新背景的像素越来越多，而旧背景和旧前景都变得越来越少。注意，这次我们是在帧循环内部创建图片kid。我们想让每一帧的效果都是新的，swapBack函数基于不同的阈值计算了不同的画面。

13.2 使用视频源

之前讲过，实时处理真实视频是很难的。这一次我们打算“作弊”：将视频保存为一列JPEG图像，处理这些JPEG图像，然后再把它们转换回一段电影。这样我们就可以将视频用作素材（比如用作背景图像）。

为处理已有的电影，我们必须把它们分解成一帧帧图像。对于MPEG电影，MediaTools应用程序可帮你完成这一任务（如图13.8所示）。使用MediaTools应用的Menu按钮可以将任何MPEG电影保存为一列JPEG图片帧。对QuickTime和AVI电影来说，

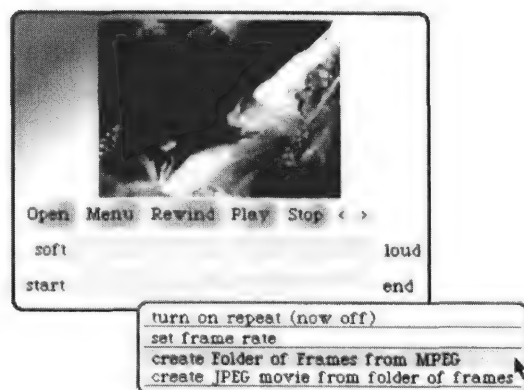


图13.8 在MediaTools应用中将一段MPEG电影保存为一系列帧画面

使用Apple QuickTime Pro这样的工具也可以达到同样的目的。

视频处理示例

mediasources目录中有一小段我们的女儿Katie跳舞的电影。下面我们来制作一段妈妈(Barb)观看女儿跳舞的电影——只是将Barb的头像组合到Katie跳舞的帧画面中(如图13.9所示)。

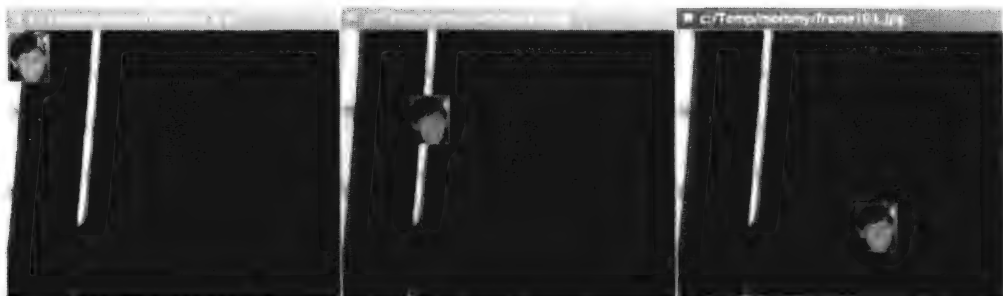


图13.9 “妈妈观看Katie跳舞”电影中的几帧画面



程序126：制作妈妈观看Katie跳舞的电影

```
import os

def mommyWatching(directory):
    kidDir=r"C://ip-book//mediasources//kid-in-bg-seq"
    barbF=getMediaPath("barbaraS.jpg")
    barb = makePicture(barbF)
    face = clip(barb,22,9,93,97)
    num = 0
    for file in os.listdir(kidDir):
        if file.endswith(".jpg"):
            num = num + 1
    printNow("Frame number: "+str(num))
    framePic = makePicture(kidDir+"//"+file)
    # 现在执行实际复制
    copy(face,framePic,num*3,num*3)
    # 现在输出一帧画面
    writeFrame(num,directory,framePic)
```

程序原理

视频源图像存储在C:/ip-book/mediasources/kid-in-bg-seq/目录中。我们把这一目录存放到一个变量中并从目录中取得Barb的照片。然后，我们使用clip方法制作一张只包含Barb脸庞的图片。帧号num随着循环不断增加，因为我们想在kidDir(存储小孩视频帧的目录)上执行os.listdir，把每一帧都作为视频素材分别读取进来。运气不错，os.listdir以字母表的次序返回各帧图片，由于文件名中的0前缀，这正好也是数字的次序。读取文件，确保它是一帧JPEG画面，然后打开它并把Barb的头像复制上去。最后，我们用writeFrame输出了一帧画面。

我们当然可以做些更高级的图像处理而不只是简单的图像组合或日落效果。比如，我们可以在电影画面上运用色键(chromakey)技术。在真实电影中，许多计算机特技都是这么做出来的。为尝试这一做法。我们拍摄了一段三个孩子(Matthew、Katie和Jenny)在蓝色屏幕前慢慢往前爬的视频(如图13.10所示)。我们的灯光有点问题，导致背景成了黑的而不是蓝的。事实证明这是个严重的错误。结果，色键处理同时改变了Matthew的裤子、Katie的头发和

Jenny的眼睛，于是你可以直接穿过他们看到月亮（如图13.11所示）。黑色与红色一样，不适合做色键背景。

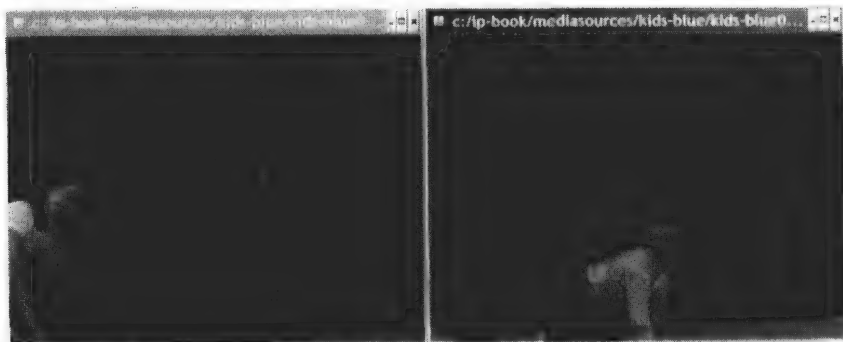


图13.10 孩子们在蓝色屏幕前爬行的几帧原始视频画面



图13.11 孩子们在月球上爬行的视频画面



程序127：使用色键技术把孩子们放到月球上去

```
import os

def kidsOnMoon(directory):
    kids="C://ip-book//mediasources//kids-blue"
    moon=getMediaPath("moon-surface.jpg")
    back=makePicture(moon)
    num = 0
    for frameFile in os.listdir(kids):
        num = num + 1
        printNow("Frame: "+str(num))
        if frameFile.endswith(".jpg"):
            frame=makePicture(kids+"//"+frameFile)
            for p in getPixels(frame):
                if distance(getColor(p),black) <= 100:
                    setColor(p,getColor(getPixel(back,getX(p),getY(p))))
            writeFrame(num,directory,frame)
```

Mark从水下拍摄了鱼的视频。水能滤掉一些红色光和黄色光，因而视频看起来蓝色较深（如图13.12所示）。我们来把视频中的红色和绿色增加一些（黄色是红色光和绿色光的混合）。我们还将创建一个新的函数，它会把图片中的红色和绿色值乘以某个输入的因子。结果如图13.13所示。



程序128：修正水下电影

```
import os

def changeRedAndGreen(pict, redFactor, greenFactor):
    for p in getPixels(pict):
        setRed(p, int(getRed(p) * redFactor))
        setGreen(p, int(getGreen(p) * greenFactor))

def fixUnderwater(directory):
    num = 0
    dir="C://ip-book//mediasources//fish"
    for frameFile in os.listdir(dir):
        num = num + 1
        printNow("Frame: "+str(num))
        if frameFile.endswith(".jpg"):
            frame=makePicture(dir+"/"+frameFile)
            changeRedAndGreen(frame, 2.0, 1.5)
            writeFrame(num, directory, frame)
```



图13.12 蓝色太深的几帧水下电影画面

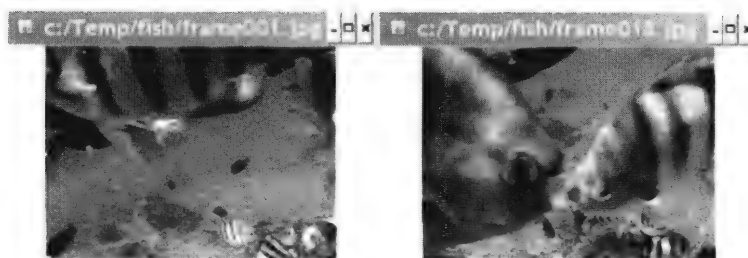


图13.13 蓝色稍轻的几帧电影画面

13.3 自底向上制作视频效果

你可能在电视广告中见过演员使用荧光棒或闪光灯在空气中“画”出东西，线条悬在空气中，就像画在白板上一样。他们是如何做到的呢？基于已有的关于图像和视频处理的知识，我们也能做出这种效果。下面我们来编写程序模拟这一过程，而且我们将按照自底向上的过程实现，以示范这种构建程序的方法。

图13.14显示了某人用荧光棒在空中画东西的4帧电影画面。这段电影我们是在天黑时拍摄的，这样荧光棒可以跟画面的其他部分形成强烈对比。怎样才能让荧光棒的明亮像素显示在后续所有画面中呢？在制作出的电影中，我们想让最后一帧就像图13.15那样。我们知道，颜色的明亮程度可以用亮度（luminance）来衡量。我们可以把一帧画面中所有颜色高于某亮度级的像素全部复制到下一帧画面中。只需针对所有画面重复这一过程，所有高亮像素都将被收集起来。



图13.14 源电影中的画面，荧光棒在空中画东西

在自底向上过程中，我们从标准库开始收集自己需要的程序片段，或者编写我们认为自己需要的程序片段。显然，在某个点上，我们必须实现亮度计算。



程序129：用于合并明亮像素的luminance函数

```
def luminance(apixel):
    return (getRed(apixel)+getGreen(apixel)+getBlue(apixel))/3.0
```

随着开发的进行，我们还应当测试一下自己编写的程序片段。luminance真的完成了我们想要的功能吗？我们来构造一个像素，把它的颜色设成某些极端的值，看看这一函数能否返回我们期望的结果。

```
>>> pict = makeEmptyPicture(1,1)
>>> pixel=getPixelAt(pict,0,0)
>>> white
Color(255, 255, 255)
>>> setColor(pixel,white)
>>> luminance(pixel)
255.0
>>> black
Color(0, 0, 0)
```

```
>>> setColor(pixel,black)
>>> luminance(pixel)
0.0
```



图13.15 目标电影的最后一帧，可以看出荧光棒的轨迹

下一步，我们需要一段能判断某个像素是否足够明亮的代码，其中需要一个阈值作为比较的基准。或许你能想到两种实现**brightPixel**的方式：传入一个阈值或使用某个默认的阈值。Python提供了一种方法，使用它我们可以同时实现这两种方式：那就是关键字参数（keyword arguments），有时也称为可选参数（optional arguments）。指定函数参数时，我们让它带上一个默认值。



程序130：用于合并明亮像素的**brightPixel**函数

```
def brightPixel(apixel, threshold=100):
    if luminance(apixel) > threshold:
        return true
    return false
```

程序原理

brightPixel接受一个像素和一个可选的阈值作为输入，阈值默认为100。如果像素的亮度大于阈值，那么它返回真。否则，程序执行到最后一行并返回假。我们可以利用已经写好的**luminance**函数来构建**brightPixel**函数。

```
>>> red
Color(255, 0, 0)
>>> setColor(pixel,red)
>>> luminance(pixel)
85.0
>>> brightPixel(pixel)
0
>>> brightPixel(pixel,80)
1
```

```

>>> brightPixel(pixel, threshold=80)
1
>>> setColor(pixel, white)
>>> brightPixel(pixel, threshold=80)
1
>>> brightPixel(pixel)
1
>>> setColor(pixel, black)
>>> brightPixel(pixel, threshold=80)
0
>>> brightPixel(pixel)
0

```

还需要什么函数呢？可以看看我们的问题陈述中是怎么说的。我们需要获得由某个目录中的所有文件组成的列表。给定一个文件列表，我们需要得到第一个文件`firstFile`，并将它与其他文件`restFiles`区别对待，这样我们就可以取出第一个文件，然后再选出其他文件中的第一个文件，并把前一个文件中的明亮像素复制到后一个文件中。然后不断重复这一过程。现在，我们把这些代码片段写下来。



程序131：用于合并像素的文件列表处理函数

```

import os

def allFiles(fromDir):
    listFiles = os.listdir(fromDir)
    listFiles.sort()
    return listFiles

def firstFile(filelist):
    return filelist[0]

def restFiles(filelist):
    return filelist[1:]

```

程序原理

这三种代码我们之前都见过，所以我們是在基于其他元素构建程序。这一次，我们只是为它们取了适当的名字并将它们改成了参数化的函数。`allFiles`直接返回由全部文件组成的列表，然后将列表排序（确保文件名以升序排列）并返回。列表中的第[0]个项目就是“第一个文件”，而[1:]则是从第二个文件开始直到列表末尾的所有文件。`firstFile`和`restFiles`就是用这种方法分别提供了我们想要的文件。我们应该把这些代码测一下，确保它们实现了预期功能。

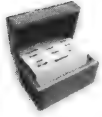
```

>>> files = allFiles("/")
>>> files
['Recycled', '_314109_', 'bin', 'boot', 'cdrom',
'dev', 'etc', 'home', 'initrd', 'initrd.img',
'initrd.img.old', 'lib', 'lost+found', 'media',
'mnt', 'opt', 'proc', 'root', 'sbin', 'srv', 'sys',
'tmp', 'usr', 'var', 'vmlinuz', 'vmlinuz.old']
>>> firstFile(files)
'Recycled'
>>> restFiles(files)
['_314109_', 'bin', 'boot', 'cdrom', 'dev', 'etc',
'home', 'initrd', 'initrd.img', 'initrd.img.old',
'lib', 'lost+found', 'media', 'mnt', 'opt', 'proc',

```

```
'root', 'sbin', 'srv', 'sys', 'tmp', 'usr', 'var',
'vmlinuz', 'vmlinuz.old']
```

构建并测试了所有独立代码片段，现在我们就可以基于这些小函数编写顶层函数了。



程序132：将明亮像素合并到新的电影中

```
def brightCombine(fromDir, target):
    fileList = allFiles(fromDir)
    fromPictFile = firstFile(fileList)
    fromPict = makePicture(fromDir+fromPictFile)
    for toPictFile in restFiles(fileList):
        printNow(toPictFile)
        # 将fromPict中的所有高亮颜色复制到toPict
        toPict = makePicture(fromDir+toPictFile)
        for p in getPixels(fromPict):
            if brightPixel(p):
                c = getColor(p)
                setColor(getPixel(toPict, getX(p), getY(p)), c)
        writePictureTo(toPict, target+toPictFile)
    fromPict = toPict
```

程序原理

`brightCombine`接受两个目录作为输入：函数从一个目录中获得移动亮点像素的帧画面，把复制了亮点像素的帧输出到另一个目录中。我们取得画面的列表并用列表中的第一帧构造一幅图片（`fromPict`）。现在，`toPictFile`将依次表示剩下的帧文件名，每次一个。而且，我们将用`toPictFile`构造一幅图片保存于变量`toPict`中。我们检查`fromPict`中的所有像素，足够亮的那些全都写到`toPict`中。然后我们让目标图片变成源图片：`fromPict = toPict`，并转向下一帧。于是，我们带着所有的亮点像素不断向前复制。

当然，我们应该测试一下，但我们把这一任务留给读者去尝试。在这一过程中，你看到的是如何构建各个元素，测试它们，然后如何基于这些元素构建更上层的东西。最终的顶层函数可能跟自顶向下设计过程中的结果完全一致。在自底向上过程中，对于目标，我们的想法开始时可能没那么清晰（这个例子中我们倒是有一个很新楚的想法），我们从构造较小的元素开始，然后逐渐往上构造更大的元素。

习题

- 13.1 一部长度为2小时、每秒60帧、画面尺寸为1 024×768像素的电影一共需要多少帧？如果直接保存每个像素的颜色，这部电影需要多少磁盘空间？别忘了每个像素需要24位来保存它的颜色。
- 13.2 什么是AVI？什么是QuickTime？什么是MPEG？视觉暂留是什么意思？什么是基于帧的动画？
- 13.3 制作一段电影，画面一帧一帧慢慢变成深褐色调。
- 13.4 编写一个函数，接受一幅图片，然后制作一段电影，电影中图片缓慢变成它的反色。
- 13.5 制作一段新电影，画面中有两个方块，每一帧当中每个方块在每个方向上都随机移动一段距离（-5~5之间）。
- 13.6 编写一个新版本的lineDetect函数（程序43），接受一个阈值，然后用lineDetect创建

一段电影，其中阈值随着帧号变化。

- 13.7 创建一段新电影，画面中有小孩在沙滩上缓缓爬行。
- 13.8 创建一段新电影，画面中Mark的脸在看Katie跳舞。
- 13.9 编写函数创建一段电影，画面中有一个物体自上往下移动，另一个物体自下往上移动。
- 13.10 编写函数创建一段电影，画面中有一个物体自左向右移动，另一个物体自右向左移动。
- 13.11 编写函数创建一段电影，画面中有一个物体沿对角线从左上角往右下角移动，另一个物体自右向左移动。
- 13.12 编写函数创建一段电影，画面中有一个实心圆出现在背景中的随机位置，然后放大后又缩小。
- 13.13 编写一个函数，在沙滩图片的不同位置画上太阳，使它看上去像太阳在一天中的移动过程。
- 13.14 编写函数创建一段电影，画面中有一段文本，开始字体很大，且位于电影底部，然后文本缓慢向着顶部移动，且每移动一次都变得更小。你可以用`makeStyle(family, type, size)`创建文本样式。
- 13.15 拍摄一段朋友在绿色屏幕前跳舞的电影。使用色键技术处理一下，使它看上去像是朋友在沙滩上跳舞。
- 13.16 到某个地方拍摄一段静止的景物画面，再到绿色屏幕前拍摄一段运动镜头，使用色键技术把两部电影融合到一起。
- 13.17 构造一段长度至少为3秒钟的动画（30帧，10fps；或者75帧，25fps）。在动画序列中至少要有三样东西在运动，其中至少有一幅合成的图像（一幅JPEG图像经必要的伸缩后复制到画面图像中）和一幅绘制的图像（矩形、直线、文本、椭圆、弧线——随便你画什么）。至少要有一件运动物体在动画行进到一半的时候改变速度——改变方向或速率。
- 13.18 <http://abcnews.go.com>是一家很受欢迎的新闻站点。我们基于它来制作一段电影。编写一个函数，接受以字符串表示的一个目录，然后：
 - 访问<http://abcnews.go.com>，提取前三条新闻标题。（提示：对于新闻报道的标题，锚都以“<a href="/wire/"开头，找到这一标签，查找锚的开头“<a href="/wire/"，这样就可以找到锚中的文本，那就是新闻标题。）
 - 在640×480的画布上创建一段纸带电影，把三条报道的标题全部显示在画面中，一条显示在y=100的位置上，另一条显示在y=200的位置上，第三条显示在y=300的位置上。产生100帧画面，两帧画面之间纸带移动的距离不超过5个像素。（经过100帧的画面，文字不至于全部移出画布——这样挺好的。）将画面保存到文件中，放在输入目录下。
- 13.19 创建一段电影，其中有一个慢慢淡出画面。你可以基于`slowFadeout`函数（程序125）来实现它。
- 13.20 还记得程序44中的图片融合吗？试着用一幅图片融合到另一幅图片中的过程制作一部电影。缓慢地增加第二幅图片的百分比（淡入），同时逐渐减少原图像的百分比（淡出）。
- 13.21 本章跟踪明亮像素的示例使用了`mediasources`目录下的`paint1`文件夹。`paint2`中还有另外一个例子，运行一下试试。
- 13.22 在跟踪明亮像素的例子中，最终的画面有点昏暗。使用`makeLighter`函数把不断复制的

像素调亮。

- 13.23 跟踪明亮像素的函数只能工作于黑暗之中吗？自己拍摄一段某人在日常亮度下使用荧光棒或闪光灯写东西的电影。函数还有效果吗？是否需要使用不同阈值呢？是否必须用不同的方式来确定“明亮”像素呢（比如把一个像素跟它周围的像素做亮度比较）？

计算机科学议题

第14章 速度

第15章 函数式编程

第16章 面向对象编程

速度

本章学习目标

- 基于对机器语言和计算机工作原理的理解选择使用编译型语言或解释型语言。
- 基于复杂性了解算法的分类，避免使用难解型算法。
- 基于对时钟频率的理解考虑如何选择处理器。
- 涉及速度优化时，能对计算机存储系统的配置做出决策。

14.1 关注计算机科学

到目前，你照着本书的讲解做了这么多事情，关于这些事情肯定有许多疑问。比如：

- 为什么Photoshop中的任何功能都比我们在JES中实现的快很多？
- 我们的程序能运行多快？
- 编写程序一定要花这么长时间吗？能否用更小的程序完成同样的功能？编写程序还能再简单一些吗？
- 用其他编程语言编写的程序又是什么样子呢？

诸如此类的问题中，大多数的答案都是计算机科学的内容或研究课题。本书这部分内容便是对这些主题的介绍，我们希望这些内容能在你进一步探索计算机科学的旅程中起到路标的作用。

14.2 什么使程序速度更快

速度到底来自哪里？你买来一台够快的计算机，在上面运行Photoshop也确实够快，拖动控件滑块颜色立即改变。而JES中同样功能的程序却会永远运行下去（或者不是“永远”，而是30秒，看哪个先到吧）。这是为什么呢？

14.2.1 什么是计算机真正理解的

实际上，计算机根本不理解Python、Java或其他任何高级语言。从基础上讲，计算机只理解一种语言——**机器语言**（machine language）。机器语言的指令只是内存中用字节存储的值，它们指示计算机完成非常底层的动作。从真正意义上讲，计算机甚至也不“理解”机器语言，它只是一台具有大量开关的机器，这些开关能够使数据按不同的方式流动。机器语言只是一组开关的设置，这些设置能引起机器中其他开关的改变。我们将这些数据开关解释（interpret）成加、减、加载数据或存储数据等不同指令。

每种计算机都可以有自己的机器语言。Windows计算机上不能运行老的Apple计算机程序，而且这不是因为任何哲学概念或市场策略方面的不同，而是因为每一种计算机都有自己专用的**处理器**（实际执行机器语言的计算机核心部件）。不同的处理器本来就无法相互理解。这就是

为什么Windows上的.EXE程序不能运行在老的Macintosh机器上，而Macintosh应用也不能运行在Windows机器上。可执行文件（几乎）永远是机器语言程序。

机器语言看起来就像一串数字——对于用户阅读，它不是特别“友好”。汇编语言（assembly language）是方便人类阅读的一组单词（或接近于单词的表示），它与机器语言一一对应。机器语言指令指示计算机完成如下一类动作：将数值存入特定内存位置或计算机中的其他特殊位置（如变量或寄存器），用于判断相等或比较的数值测试，将数值相加或相减。

下面的例子是把两个数值相加，再把结果保存到某个位置的一种汇编程序（以及由汇编器产生的相应机器语言程序）：

```
LOAD #10, R0      ; 数值10载入特殊变量R0
LOAD #12, R1      ; 数值12载入特殊变量R1
SUM R0, R1        ; 特殊变量R0与R1相加
STOR R1, #45      ; 结果存入45#内存位置
```

```
01 00 10
01 01 12
02 00 01
03 01 45
```

下面的汇编程序可以根据条件做出决策：

```
LOAD R1, #65536    ; 从键盘取得一个字符
TEST R1, #13       ; 它是ASCII码为13的键（回车）吗？
JUMPTTRUE #32768   ; 如果是，跳转到程序的另一部分
CALL #16384        ; 如果不是，调用函数处理新行
```

对应的机器语言：

```
05 01 255 255
10 01 13
20 127 255
122 63 255
```

对计算机而言，输入和输出设备常常只是内存位置。或许，在你往65 542这个位置存入数值255的时候，屏幕上（101，345）处像素的红色分量便突然变成了最大强度。或许，每次计算机从内存位置897 784处读取信息时，读到的都是刚刚从麦克风传来的最新样本值。通过这种方式，这些简单的存取指令就能处理多媒体。

机器语言执行速度非常快。Mark在一台处理器频率为900 MHz的计算机上录入了本章的初稿。900 MHz的精确含义不好定义，基本上它意味着这台计算机每秒钟能处理90亿条机器语言指令。类似地，一颗2 GHz的处理器每秒钟能处理20亿（2 billion）条指令。一个12字节长的机器语言程序，大约对应 $a=b+c$ 这样的计算，在Mark的这种中档计算机上执行约需12/900 000 000秒。

14.2.2 编译器和解释器

像Adobe Photoshop和Microsoft Word这样的应用程序通常是编译型的程序。换句话说，它

们是由C或者C++这样的计算机语言编写的，然后使用一个称为编译器（compiler）的程序翻译成了机器语言。此后程序便以处理器的基础频率运行。

而像Python、Java、Scheme、Squeak、Director和Flash这样的编程语言（大多数情况下）是解释型的。它们执行的速度要慢一些。两者的区别是就先翻译后执行与直接执行指令之间的区别。

下面详细的例子可能更有帮助。考虑下面的练习：

编写一个函数doGraphics，接受一个列表作为输入。doGraphics函数首先基于mediasources文件夹下的640×480.JPG文件创建一张画布。你将根据输入列表中的命令在画布上作图。

列表中的各个元素将是命令字符串，这些字符串分为两类：

- “b 200 120”表示在x=200，y=120的位置画一个黑点。数字当然会变，但命令永远是“b”。你可以假定输入的坐标都是三位数。
- “l 000 010 100 200”表示从（0，10）到（100，200）画一条直线。

输入列表可以是：["b 100 200", "b 101 200", "b 102 200", "l 102 200 102 300 102 300 200 300"]。（元素数目可以是任意的）。

以下是这个练习的一种解法：我们查看列表中的每个字符串，比较第一个字符，看它是“黑色像素”命令还是“列表”命令。然后提取相应的坐标（使用int()转换成数字）并执行相应的绘图命令。这种解法是有效的——见图14.1。

```
>>> canvas=doGraphics(["b 100
200","b 101 200","b 102
200","l 102 200 102 300","l
102 300 200 300"])
Drawing pixel at 100 : 200
Drawing pixel at 101 : 200
Drawing pixel at 102 : 200
Drawing line at 102 200 102 300
Drawing line at 102 300 200 300
>>> show(canvas)
```

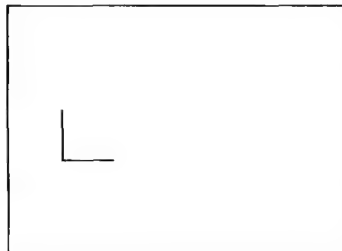


图14.1 运行doGraphics解释器



程序133：解释执行程序中的绘图命令

```
def doGraphics(mylist):
    canvas = makePicture(getMediaPath("640 x 480.jpg"))
    for command in mylist:
        if command[0] == "b":
            x = int(command[2:5])
            y = int(command[6:9])
            print "Drawing pixel at ",x,":",y
            setColor(getPixel(canvas, x,y),black)
        if command[0] == "l":
            x1 = int(command[2:5])
            y1 = int(command[6:9])
            x2 = int(command[10:13])
            y2 = int(command[14:17])
            print "Drawing line at",x1,y1,x2,y2
            addLine(canvas, x1, y1, x2, y2)
    return canvas
```

程序原理

我们用输入参数mylist接受了绘图命令的列表。然后打开空白的640×480.JPG画面准备画图。针对输入列表中的每个字符串command，我们检查第一个字符（command[0]），判断它是哪种命令。如果是“b”（黑色像素），我们便从字符串中提取x和y坐标（由于这些数字串的长度都一样，我们自然知道每个坐标的精确位置），然后在画布上绘出这个像素。如果是“l”（直线），则取得4个坐标值并画直线。最后返回画布。

我们刚刚实现的是一种新的绘图语言，我们甚至创建了读取新语言指令的解释器，并基于这些指令画出了图片。从原理上讲，这正是Postscript、PDF、Flash和AutoCAD所做的事情。在这类软件所用的文件格式中，描述图片的方式与我们的绘图语言完全一样。需要在屏幕上呈现图像时，软件便会解释文件中的命令。

这么小的例子或许不足以说明问题，但相对来讲，这确实是一种速度较慢的语言。考虑下面的程序——比起读取列表并解释之的做法，它会快一些吗？这个程序与图14.1中的命令列表产生的图形完全一样。

```
def doGraphics():
    canvas = makePicture(getMediaPath("640 x 480.jpg"))
    setColor(getPixel(canvas, 100,200),black)
    setColor(getPixel(canvas, 101,200),black)
    setColor(getPixel(canvas, 102,200),black)
    addLine(canvas, 102,200,102,300)
    addLine(canvas, 102,300,200,300)
    show(canvas)
    return canvas
```

一般情况下，我们可以（正确地）猜出：上面给出的直接指令运行起来要比读取列表并解析的速度快一些。或许打个比方更有助于理解：Mark在大学里选修了法语。但他说自己学得很糟糕。假如有人用法语给他写了一份指令列表。他可以慢慢查出每个单词的意思，弄清每条指令的意思，然后完成这条指令。如果让他把这列指令再做一遍，他会怎么办呢？他会把单词再查一遍。如果做10遍呢？那就查10遍。现在，假设他写下了这些法语指令的英语译文（英语是他的母语）。这样一来，他就能快速地重做这列指令，做多少遍都无所谓，他不需要花任何时间去查单词。一般来说，理解语言需占用时间，而这些时间是额外的开销——直接执行指令（或画图）总会快一些。

下面又是一种思路：我们能否产生上面的程序呢？能否编写一个程序，基于我们发明的图形语言，接受一个命令列表作为输入，然后输出一个Python程序，用它画出同样的图片？事实证明，编写这样一个程序没有你想的那么难。这样的程序就是图形语言的编译器了。



程序134：新图形语言的编译器

```
def makeGraphics(mylist):
    file = open("graphics.py","wt")
    file.write('def doGraphics():\n')
    file.write('    canvas = makePicture(getMediaPath ("640 x 480.jpg"))\n');
    for i in mylist:
        if i[0] == "b":
            x = int(i[2:5])
            y = int(i[6:9])
            print "Drawing pixel at ",x,":",y
            file.write('    setColor(getPixel(canvas, '+str(x)+' , '+str(y)+'),-
```

```
        black)\n')
    if i[0] == "l":
```

~这几行程序应该与下面的一行连续。Python中一条命令不可以跨多行。

```
    x1 = int(i[2:5])
    y1 = int(i[6:9])
    x2 = int(i[10:13])
    y2 = int(i[14:17])
    print "Drawing line at",x1,y1,x2,y2
    file.write(' addLine(canvas, '+str(x1)+'+', '+str(y1)+'+', '+ str(x2)+'+', '~
    '+str(y2)+'')\n')
file.write(' show(canvas)\n')
file.write(' return canvas\n')
file.close()
```

~这几行程序应该与下面的一行连续。Python中一条命令不可以跨多行。

程序原理

编译器接受的输入与解释器是一样的，但它打开的不是一张用于画图的画布，而是一个文件（file）。我们向文件写入了doGraphics函数的开头——def语句以及用于创建画面（canvas）的代码（缩进两个空格，使它们处于doGraphics函数的语句块中）。注意，在这个程序中，并未真正生成画布——只是写出了用于生成画布的命令，这些命令将在以后执行doGraphics时得以执行。然后与解释器一样，判断绘图命令是什么（“b”还是“l”）并从输入字符串中找出坐标。然后，向文件中输出了用于画图的命令。最后，输出了显示（show）并返回画布的命令并关闭了文件。

现在，编译器引入了一大堆额外开销。我们仍需查询各个命令的意思。如果只是运行一个小小的绘图程序，且只运行一次，我们完全可以使用解释器。但如果需要运行10次或100次呢？这时我们只付出一次编译程序的开销，剩下的9次或99次，程序运行起来要多快有多快。比起100次解释程序的开销，总体上这样更快，那几乎是肯定的。

编译器就是这么个东西。Photoshop和Word这样的应用程序都是用C或C++语言编写，然后编译成了等价的机器语言程序。机器语言程序完成的功能与C语言描述的完全一样，正如我们的编译器创建的绘图程序与我们的图形程序所描述的完全一样。但机器语言程序运行起来比解释C或C++程序要快多了（如果C或C++可以解释执行的话）。

Jython程序实际上不只解释一次，而是两次。Jython是用Java编写的，而Java程序通常并不编译成机器语言。（Java可以编译成机器语言——只是一般人们不会那样使用它。）Java程序被编译成一种特殊的“机器语言”，这种机器语言针对一种假想的处理器，即虚拟机（virtual machine）。Java虚拟机不是真正存的物理处理器，而只是一套处理器的定义。它有什么好处呢？事实证明，由于机器语言非常简单，构造一种机器语言的解释器也是非常容易的事情。

结果，人们很容易做出可运行于各种机器上的Java虚拟机解释器。这意味着用Java编写的程序可以一次编译到处运行。大到大型计算机，小到手表这样的设备，都可以运行同样的Java程序。

当你在JES中运行程序的时候，程序实际被编译成了Java——一个等价的Java程序会产生出来。然后JES针对Java虚拟机编译这个Java程序。最后，由Java虚拟机解释器运行从你的程

序一路编译过来的Java机器语言。与直接运行同一程序的某种编译形式相比，所有这一切都使它速度更慢。

关于“为什么Photoshop总是比JES快”？以上就是问题答案的第一部分。在JES中，程序被解释了两次，因此肯定比直接以机器语言运行的Photoshop慢多了。

那为什么要有解释器呢？充分的理由有很多，以下是其中的三个：

- 你喜欢命令区吗？有没有直接在命令区中输入示例代码，只为了试验一下的经历？这种交互式、解释性和试验性的编程方式只有在使用解释器的情况下才能实现。编译器不会让你一行行地尝试代码并打印结果。解释器很适合学习者使用。
- 一旦程序编译成Java机器语言，它就可以原封不动地到处使用。巨型计算机上可以用，可编程的烤箱上也可以用。这对软件开发者来说是一笔可观的节省。他们只需交付一个程序，程序就可以在各种机器上运行。
- 虚拟机比机器语言更安全。以机器语言运行的程序可能做各种不安全的事情。虚拟机则可以仔细跟踪它所解释的程序，确保它只做安全的事情。

14.2.3 什么限制了计算机的速度

关于Photoshop为什么会更快，编译型程序相对于解释型程序的先天优势只是答案的一部分。更深层的部分，而且实际上能使解释型程序比编译型程序还要快的部分，在于算法的设计。或许有人会想：“噢，现在慢点儿不要紧。等上18个月，处理器速度就会加倍，到时候就不慢了。”然而，有些算法会慢到在你的有生之年都不会结束，还有一些算法压根写不出来。重写算法，以更聪明的方式描述我们想让计算机执行的任务，可以对性能产生显著的影响。

算法 (algorithm) 是计算机为解决一个问题而必须遵循的行为方式的描述。各种算法翻译成可执行的形式就构成了程序 (Python中是函数)。同样的算法可通过多种语言实现。解决同样的问题也常常有多种算法——有些计算机科学家对算法进行了研究，提出一些比较它们的方法来判定哪个算法更好。

我们曾见过好几种算法，它们以不同的形式出现，实际完成的功能却是一样的。比如：

- 通过采样来缩放图片的尺寸或增减声音的频率。
- 通过混合来合并两张图片或两段声音。
- 对声音或图片做镜像操作。

比较算法可基于多种准则。一种准则是算法运行时需要的空间数量，即算法需要多少内存。对媒体计算来说，这会成为重要问题，因为保存所有的媒体数据需要大量内存。想象一下，如果某个算法需要把一段电影的所有画面同时安放在内存中的一个列表里面，那是多么糟糕的事情。

最常用的比较算法的准则是时间，即运行算法所需的时间长度。我们指的不是时钟时间，而是算法需要多少步骤。计算机科学家使用大O标记 (Big-O Notation)，或者 $O()$ 来表示运行算法所需的时间量级。大O的思想是表达程序随着输入数据规模的增长会变慢到什么程度。它尽量忽略语言之间的差别，甚至编译型程序和解释型程序之间的差别，而专注于执行程序所需的步骤数量。

考虑一下`increaseRed()`和`increaseVolume()`这种基本的图片处理和声音处理函数。这类函数的某些复杂度隐藏在了`getPixels()`和`getSamples()`等函数中。通常，我们仍然认为这些函数的复杂度为 $O(n)$ ，即运行程序需要的时间与数据规模之间呈线性的比例关系。如果图片的

尺寸或声音的长度加倍，可以想象，运行程序所需的时间也会加倍。

计算大O的时候，我们通常把循环体并为一步，认为这些函数将每个样本或像素处理一次，于是，这些函数中真正耗费时间的因素就成了循环本身，循环内的语句数目倒是关系不大了。

除非循环体中还有另一层循环，否则它就是 $O(n)$ 了。在时间上，循环有乘法效应。嵌套的循环中，各循环体所需的时间将成倍累积起来。考虑如下的玩具程序：

```
def loops():
    count = 0
    for x in range(1,5):
        for y in range(1,3):
            count = count + 1
            print x,y,"--Ran it ",count,"times"
```

可以看到，运行时它实际执行了8次——4次 x ，2次 y ， $4 \times 2 = 8$ 。

```
>>> loops()
1 1 --Ran it 1 times
1 2 --Ran it 2 times
2 1 --Ran it 3 times
2 2 --Ran it 4 times
3 1 --Ran it 5 times
3 2 --Ran it 6 times
4 1 --Ran it 7 times
4 2 --Ran it 8 times
```

处理电影的代码又是什么情况呢？需要的时间那么长，那它是不是更复杂的算法呢？实际上不是这样。电影代码同样是每个像素只处理一次，因此，仍然是 $O(n)$ 的，只不过这里的 n 实在太大了！

然而，并非所有算法都是 $O(n)$ 的。有一类算法叫做**排序算法**，用于把数据排列成字母表顺序或数字顺序。其中有一种简单的称为**冒泡排序** (bubble sort)，它的复杂度为 $O(n^2)$ 。在冒泡排序中，我们循环遍历列表中的元素，比较相邻的两个，如果次序不对就交换一下。我们需要持续执行这一过程，直到一轮遍历下来未发生任何交换动作，即数据已经有序为止。

举个例子，如果开始的时候列表是 (3, 2, 1)，以下步骤显示了冒泡排序过程中列表的变化：

(3, 2, 1)	# 比较3和2并交换次序
(2, 3, 1)	# 比较3和1并交换次序
(2, 1, 3)	# 比较2和1并交换次序
(1, 2, 3)	# 没有交换，列表已经有序

如果列表中有100个元素，那么使用这种算法为元素排序所需的步骤数量就是万级的。然而，有一些更聪明的算法（比如**快速排序**），复杂度只有 $O(n \log n)$ 。在快速排序中，我们从待排序列表中选择一个值作为基准点 (pivot)。然后，原始列表中的值被划分到两个列表中：所有小于基准点的值移到第一个列表中，而大于或等于基准点的值移到另一个列表中。然后，针对两个新的列表，我们再次使用快速排序算法来排序。单元素列表本来就是有序的，所以如果某个列表小到只包含一个元素，算法就直接返回列表本身。

(5 1 3 2 7)	# 选出3作为基准点
(1 2) 3 (5 7)	# 选出2和7作为基准点
(1) 2 3 (5) 7	# 所有列表长度为1，于是直接返回

(1 2 3 5 7)

合并所有返回的列表

同样含有100个元素的列表使用快速排序处理只需460步。当需要处理10 000个客户的信息时，这种差距就会带来明显的物理时间上的差别了。

14.2.4 让查找更快

考虑如何用字典查单词。一种方法是先在第一页找，然后看下一页，然后再看下一页……这称为线性查找，复杂度为 $O(n)$ 。这种方法效率不高。最好情况（执行算法最快的一种可能）下问题可以在一步之内解决——单词出现在第一页的情况。最坏情况下则需要 n 步，其中 n 为字典的页数——单词可能出现在最后一页。平均情况是 $n/2$ 步——单词在中间一页上。

我们可以把这种方法实现为列表中的查找。



程序135：列表的线性查找

```
def findInSortedList(something, alist):
    for item in alist:
        if item == something:
            return "Found it!"
    return "Not found"

>>> findInSortedList("bear", ["apple", "bear", "cat", "dog",
                                "elephant"])
'Found it!'
>>> findInSortedList("giraffe", ["apple", "bear", "cat",
                                   "dog", "elephant"])
'Not found'
```

然而，我们可以利用字典的内容有序这一事实。这样查找单词的方法可以更聪明一些，在 $O(\log n)$ 的时间内完成（当 $2^x = n$ 时，我们说 $x = \log(n)$ ）。将字典从中间一分为二，查看中间这一页并判断待查单词在这页之前还是之后。如果是之后，则在字典中从中间到结尾的范围内查找（再次把字典分开，但这次分的是从中间到结尾的这部分）。如果是之前，则在字典中从开头到中间的部分查找（将这部分从中间一分为二）。持续这一过程，直到查到单词或者发现它不可能在字典中。这种算法更加高效，最好情况下，单词出现在第一次查看的地方。平均情况和最坏情况下，它需要 $\log(n)$ 步—— n 页字典不断一分为二，最多分 $\log(n)$ 次。

这种查找方法称为二分查找（binary search），以下是它的一种简单实现（不是最好的，但可以说明问题）。



程序136：简单的二分查找

```
def findInSortedList(something, alist):
    start = 0
    end = len(alist) - 1
    while start <= end: # 只要还有可以查找的项目
        checkpoint = int((start+end)/2.0)
        if alist[checkpoint] == something:
            return "Found it!"
        if alist[checkpoint] < something:
            start = checkpoint + 1
        if alist[checkpoint] > something:
            end = checkpoint - 1
    return "Not found"
```

程序原理

开始时，我们让低端标记start位于列表开头(0)，而end位于列表末尾(列表长度减1)。只要start小于或等于end，就一直查找。计算start和end的中点checkpoint，然后检查目标是否找到。如果找到，任务完成，返回“Found It!”。如果没找到，通过判断决定把start移至checkpoint之后一个位置，还是把end移至checkpoint之前一个位置，然后继续查找。如果能退出整个循环而没有返回“Found It!”，那我们就返回未找到目标(“Not Found”)。

要弄明白整个查找过程，可以在计算checkpoint之后添加一行代码，把checkpoint、start和end的值都打印出来：

```
printNow("Checking at: "+str(checkpoint)+"
  Start:"+str(start)+" End:"+str(end))
>>> findInSortedList("giraffe",["apple","bear","cat",
  "dog"])
Checking at: 1 Start:0 End:3
Checking at: 2 Start:2 End:3
Checking at: 3 Start:3 End:3
'Not found'
>>> findInSortedList("apple",["apple","bear","cat",
  "dog"])
Checking at: 1 Start:0 End:3
Checking at: 0 Start:0 End:0
'Found it!'
>>> findInSortedList("dog",["apple","bear","cat",
  "dog"])
Checking at: 1 Start:0 End:3
Checking at: 2 Start:2 End:3
Checking at: 3 Start:3 End:3
'Found it!'
>>> findInSortedList("bear",["apple","bear","cat",
  "dog"])
Checking at: 1 Start:0 End:3
'Found it!'
```

14.2.5 永不终止和无法编写出的算法

来一个思维实验(thought experiment)吧：假设你想编写个程序为自己产生好听的歌曲。程序把一些声音片段重新组合，这些片段是你从各种乐器中听到过的最好的乐段——总共有60段。你打算产生60段音乐的全体组合(某个乐段有时包含在结果中，有时不包含在结果中，有时出现在前面，有时出现在后面)。你还想从中挑出短于2分30秒(最适合收音机播放的时间)，而且高低音量的搭配也恰如其分(假定你已经有一个可以检查这项指标的checkSound()函数)的组合。

组合的总数有多少呢？暂且忽略次序问题。假如有三段声音：*a*、*b*和*c*，可能的歌曲是：*a*、*b*、*c*、*bc*、*ac*、*ab*和*abc*。再试一下两段声音和四段声音的情况，你会发现，模式与我们以前讨论位时一模一样：对于*n*件东西，或包含或不包含某件东西的所有组合数目为 2^n 。(实际上存在一首“空白歌曲”，若无视这一事实，那就是 $2^n - 1$ 。)

所以，我们的60段声音将组合出 2^{60} 首歌曲，需要我们一一检查长度和声音约束。 2^{60} 等于1 152 921 504 606 846 976。让我们假想一下，做一次长度和声音检查只需一条指令(是的，我反正不信，但我们可以假装相信)。那么，在1.5 GHz的计算机上，我们可以在768 614 336

秒的时间内处理完些组合。还是列一下吧：那是12 810 238分钟，或213 504小时，或8896天。也就是说，运行程序需要24年。然后，因为摩尔定律每隔18个月会把处理器速度增加一倍，所以我们很快就可以用更短的时间跑完程序。只要12年就够了！如果我们还关心组合的次序（比如 abc 、 cba 和 bac 是不一样的），那会有多少种情况呢？这个数字中光0就有63个。

从所有情况中找出绝对最优的组合永远是极其耗时的任务。对于这样的算法，类似 $O(2^n)$ 这样的时间复杂度并不罕见。但还有其他一些问题，看似可以在合理的时间内完成，实际却不是。

这些问题当中，比较著名的一个就是旅行商问题（Traveling Salesman Problem）。想象自己是一名负责很多客户的售货员——比如说客户数量是30，前面最佳歌曲问题的一半。为提高工作效率，你想在地图上找一条能把每个客户访问一次，且不会重复访问的最短路径。

要求给出旅行商问题的最优解，一种最有名的算法是 $O(n!)$ 级的。那可是 n 的阶乘。另外有些耗时较短的算法能给出近似最短，但无法保证绝对最短的路径。对30个城市来说，使用这种 $O(n!)$ 复杂度的算法需要执行30!个步骤，或者说265 252 859 812 191 058 636 308 480 000 000步。到1.5 GHz的处理器上运行看吧——在你有生之年是运行不完的。

真正严重的问题是：旅行商问题并不是人为搞出来的玩具题目。确实有人需要在全世界范围内规划最短路由。还有一些类似问题，从算法上考虑与旅行商问题如出一辙，比如规划机器人在厂房中的行走路线。这是个又大又难的问题。

计算机科学家把问题归为三大类：

- 许多问题，比如排序，可以用运行时间为多项式复杂度（比如 $O(n^2)$ ）的算法解决，我们把这类问题称为P类问题（P代表“多项式”）。
- 另一些问题，比如求最优组合，存在已知的算法，但解法太大太难，即使中等规模的数据量都难以在合理的时间内解决。我们把这类问题称为难解型（intractable）问题。
- 还有另一些问题，如旅行商问题，看似难解，但可能存在P类解法，只是我们尚未发现。我们把这类问题称为NP类问题。

理论计算机科学领域最大的未解问题之一就是证明要么NP和P完全不同（意味着我们永远不能在多项式时间内解决旅行商最短路径问题），要么P包含NP。

你可能疑惑，有关算法的问题可以“证明”吗？毕竟我们有这么多不同的编程语言和编写算法的不同方式。如何能确定地证明一件事情是可做或不可做的呢？然而，这的确可以。事实上，Alan Turing（阿兰·图灵）甚至证明了某些算法是编写不出来的。

在编写不出来的算法当中，最著名的一个是程序停止问题（Halting Problem）。我们编写过读取或输出其他程序的程序。可以想象，一个程序完全可以读取另一个程序并输出相关信息（比如此程序中有多少print语句）。那么，能否编写一个程序，输入另一个程序（比如通过文件），然后告诉我们那个程序会不会停止呢？考虑这样一种情况：输入程序中有一些复杂的while循环，导致我们难以判定while循环表达式会不会变成false。然后再想象一下这样一组循环相互嵌套的情况。

Alan Turing证明了这样的程序是编写不出来的。他用的是反证法。如果能编写出这样一个程序（命名为H），那么可以把程序本身作为它的输入。这时，H接受了输入的程序，对吧？现在，如果修改了程序H（得到H2），使得如果H说“这个程序会停止！”，则让H2做永久循环（比如改成while 1:）。Turing证明了这样一种构造将导出以下结论：程序仅在无限循环时才会停止，且仅当它声称自己无限循环时才会停止。

最让人惊叹的是：Turing是在1936年给出这一证明的——比第一台计算机的诞生早了近10

年。他定义了一种称为图灵机的数学概念上的计算机，并在物理计算机出现之前完成了这种证明。

再来一个思维实验：人类的智能是可计算的吗？我们的大脑也在执行一个过程，这才使我们能够思考，是这样吗？能把这一过程编写成算法吗？如果让计算机执行这一算法，它算是在思考吗？人可以归约为一台计算机吗？这些都是人工智能领域的大问题。

14.2.6 为什么Photoshop比JES更快

现在，我们可以回答为什么Photoshop比JES更快这一问题了。首先，Photoshop是编译型的程序，它直接以机器语言的速度运行。

但还有第二点，Photoshop使用的算法比我们使用的更聪明。作为例子，考虑一下查找颜色的程序，比如在色键处理或染红Katie头发的程序中。我们知道，背景色和头发的颜色紧挨在一起。如果直接从要找的颜色所在的位置开始查找，直到无法再找到这种颜色为止，而不是线性地查找所有像素，效果会怎样呢？通过这种方式找到边界才是更聪明的查找方法，Photoshop用的就是这样的方法。

14.3 什么使计算机速度更快

计算机一直在变得更快——摩尔定律为我们保证了这点，但这无助于比较同一摩尔定律时代的计算机。如何比较报纸上的计算机广告并确定哪一台才是真正最快的呢？

当然，速度快慢只是挑选计算机的准则之一。除此之外，还有成本问题，需要多大的磁盘空间，需要哪些扩充特性等。但在这一节，我们仅从速度方面考查计算机广告上各种指标的意义（参见图14.2中的示例）。

- | | |
|---|--|
| <ul style="list-style-type: none"> • AMD Athlon™ XP Processor 3000+ with QuantiSpeed™ Architecture • 400MHz Front Side Bus • 512KB L2 Cache • DVD-ROM Drive • CD-RW Drive • 512MB DDR SDRAM • 120.0GB Hard Drive | <ul style="list-style-type: none"> • Intel® Celeron® Processor 2.7GHz • CD-RW Drive • 400MHz Front Side Bus • 128KB L2 Cache • 256MB DDR SDRAM • 40.0GB Hard Drive |
|---|--|

图14.2 两则计算机广告中取出的示例指标

14.3.1 时钟频率和实际的计算

如果计算机广告上说他们有“某品牌处理器，2.8 GHz”或“另一品牌处理器，3.0 GHz”，他们指的是时钟频率（clock rate）。处理器是计算机的智慧所在——它是进行决策和执行计算的部件。处理器以一种特定的节奏执行所有计算。想象一下军训教官喊着“一！二！三！四！”的情形，时钟频率就是那个样子——它让你知道教官以多快的速度喊数字。2.8 GHz的时钟频率意思就是每秒28亿次时钟脉冲（教官喊28亿次数字）。

然而，这不是说处理器在每一次喊数时都完成有用的动作。有的计算需多步完成，因而，完成一次有用的计算可能需要多个时钟脉冲。但一般来讲，更快的时钟频率意味着更快的计算速度。当然，如果是同一种类的处理器，更高的时钟频率自然会带来更高的计算速度。

2.8 GHz和3.0 GHz真的有区别吗？或者换个问题，1.0 GHz的处理器X和2.0 GHz的处理器Y速度会不会一样呢？这些问题更难回答了。实际上，这无异于争论道奇汽车跟福特汽车哪种

更好。每种处理器都有其追捧者和唱衰者。有人会说，出于优良的设计，X处理器可在短短数个时钟脉冲内完成某种查找，因此，虽然它频率低，但无疑速度更快。又有人会说，整体上讲Y处理器仍然胜出，因为每次计算需要的平均时钟脉冲数很少——何况，X能做的那种查找又有多常见呢？这近乎信仰之争了。

如今，你能买到的不少多核计算机。每个核都是个完整的处理器。双核（dual core）计算机的主芯片上实际有两个处理器。四核（quad core）有四个处理器。这些计算机的速度是否比一般的快2倍或4倍呢？不幸的是，关系没有这么直接。并非所有程序在编写时都利用了多核特性。很难编写一个程序来表达这样的目标：“OK，下面一段计算可以并行执行，这一小段在这儿，那一小段在那儿，最后，我们会这样合并它们。”如果你用的软件在编写时都没有考虑利用多核，那么使用多核处理器时计算也不会快很多。今天，计算机科学的巨大挑战之一就是如何充分利用多核让计算机更快地为人们工作。

真正的答案是：你应该在考虑购买的计算机上尝试一些实际任务，实际感受一下它够不够快。也可以查看各种计算机杂志上的评论——它们经常用实际的工作（比如Excel中的排序或者Word中的翻页）来测试计算机的速度。

14.3.2 存储：什么使计算机速度慢

处理器的速度只是使计算机更快或更慢的因素之一。也许，一个更大的因素是处理器从哪里取得它所处理的数据。计算机处理图片的时候，图片在哪里？这个问题要复杂得多。

可以把计算机的存储看成一种层次式的结构，从最快的到最慢的。

- **最快的存储是高速缓冲存储器（cache memory）。**高速缓存是一种物理上与处理器位于同一硅芯片（或离它非常近）的存储器。处理器负责将尽量多的数据放入高速缓存，而且只要还需要这些数据，就让它一直放在那里。访问高速缓存比访问计算机上任何其他东西都要快得多。拥有的高速缓存越多，计算机就越能以更高的速度访问更多数据。当然，高速缓存也是计算机上最昂贵的存储部件。
- **随机访问存储器（Random Access Memory, RAM）（SDRAM或其他种类的RAM）**是计算机上的主存储器。256 MB的RAM表示它能保存256百万字节的信息。1 GB的RAM则是10亿字节的信息。RAM存储是运行时程序的栖身之所，计算机直接处理的数据也位于RAM中。信息在加载到高速缓存之前首先会存在于RAM中，RAM不像高速缓存那么昂贵，想让计算机更快，买RAM可能是最合算的投资。
- **硬盘**是存储所有文件的地方。现在计算机上正在运行的程序，之前都以.exe文件（可执行文件）的形式存在于硬盘上。所有的数字图片、数字音乐、字处理文件、电子表格文件等都存储在硬盘上。硬盘是计算机上访问速度最慢的存储部件，但同时也是容量最大的。40 GB的硬盘上可以存储400亿字节。那可是巨大的空间了——如今，这还不算大的。

在不同层级之间移动数据必然引入巨大的速度差异。曾有人说道：如果说访问高速缓存相当于从桌子上取一个纸夹，那么从硬盘上取数据就相当于去一趟离地球4光年的半人马座星。显然，我们从磁盘上取得数据的速度并不算慢（这也说明高速缓存的速度是多么惊人！），但这个比方还是强调了不同层级在访问速度方面的巨大差异。最直接的结论是：拥有越多更快的存储器，处理器获取信息的速度就越快，整体的处理性能就越高。

偶尔你会看到提及**系统总线（system bus）**的广告。系统总线决定信号在计算机中传递的方式——从视频设备到网络再到硬盘，从RAM到打印机。更快的系统总线显然会带来更快的

整体系统，却不一定影响你对（比方说）Photoshop或JES的速度体验。第一，即使最快的总线也比处理器慢很多——那可是每秒4亿次脉冲（400 MHz）与每秒40亿次脉冲（4 GHz）的差别。第二，系统总线通常不会影响对高速缓存或存储器的访问，而这些地方才是系统总体性能的得失之地。

你可以采取措施让硬盘尽可能快地服务于自己的计算任务。对处理器时间来说，磁盘的速度并没有那么重要——最快的磁盘也比最慢的RAM慢很多。在磁盘上为内存交换（swapping）留下足够的空间很重要。当计算机没有足够的RAM空间用于你所请求的任务时，它就会从RAM中把一些目前暂时不用的数据存到硬盘上。用磁盘来回移动数据是个缓慢的过程（相对缓慢，与访问RAM的速度相比）。如果有包含足够可用空间的快速磁盘，计算机就不必四处查找交换空间，这有助于提高处理速度。

网络又是什么情况呢？在速度方面，网络对你的帮助不大。网络比磁盘慢好几个数量级。不同的网络速度确实有差异，以至于影响你的整体体验，但不至于影响计算机的处理速度。有线Ethernet连接通常比无线Ethernet连接更快。调制解调器连接则慢一些。

14.3.3 显示

显示方面呢？显示速度会影响计算机的速度吗？不会的。计算机真的很快。即使是很大的显示设备，它也能快速重画，快到你感觉不到。

唯一一种让显示速度不再无关紧要的应用程序是真正高端的计算机游戏。有些游戏玩家声称他们能感觉到画面更新时每秒50帧与每秒60帧之间的差别。如果你的显示器真的很大，而且每次画面更新时都要重画整屏像素，或许一颗更快的处理器会带来可以感知的不同。但多数现代计算机的画面更新速度都已经很快，快到你根本注意不到这种差异。

习题

14.1 名词解释：

- 解释器
- 编译器
- 机器语言
- RAM
- 高速缓存
- P类问题
- NP类问题

14.2 上网找一些不同排序算法的动画。哪种算法最快？哪种最慢？

14.3 编写函数对列表执行插入排序。

14.4 编写函数对列表执行选择排序。

14.5 编写函数对列表执行冒泡排序。

14.6 编写函数对列表执行快速排序。

14.7 下面的代码将打印多少次消息？

```
for x in range(0,5):
    for y in range(0,10):
        print "I will be good"
```


14.8 下面的代码将打印多少次消息？

```
for x in range(1,5):
    for y in range(0,10,2):
        print "I will be good"
```

14.9 下面的代码将打印多少次消息？

```
for x in range(0,3):
    for y in range(1,5):
        print "I will be good"
```

14.10 `clearBlue`方法的大O复杂度是怎样的？

14.11 `lineDetect`方法的大O复杂度是怎样的？

14.12 基于以下命令单步跟踪`findInSortedList`中的二分查找算法。

```
findInSortedList("8", ["3", "5", "7", "9", "10"])
```

14.13 基于以下命令单步跟踪`findInSortedList`中的二分查找算法。

```
findInSortedList("3", ["3", "5", "7", "9", "10"])
```

14.14 基于以下命令单步跟踪`findInSortedList`中的二分查找算法。

```
findInSortedList("1", ["3", "5", "7", "9", "10"])
```

14.15 基于以下命令单步跟踪`findInSortedList`中的二分查找算法。

```
findInSortedList("7", ["3", "5", "7", "9", "10"])
```

14.16 你已经分别见过P类问题（如排序和查找）、难解型问题（如最优歌曲组合问题）和NP类问题（如旅行商问题）的例子。上网搜索一下，每一类问题至少再找出一个例子。

14.17 找一项可以在JES中长时间运行的任务（比如对大图片做色键处理），这样你可以用秒表计一下运行时间。然后，到不同内存数量、不同时钟频率（甚至不同的高速缓存数量，如果你能找到的话）的多台计算机上运行同一项JES任务并计时。看看不同的因素对完成这项JES任务所需的时间会带来哪些影响。

14.18 除了证明停机问题是不可解的外，Alan Turing还因为计算机科学领域的另外一项发现而闻名于世。他提供了一种证明计算机是否真正达到智能化的测试方法。这种测试方法的名字叫什么？又是如何测试的？你是否认同这是对智能的测试？

14.19 有些问题存在找到最优解的算法，但需要的时间太长，人们如何获得这类问题的答案呢？有时人们会使用启发式方法（heuristics），即一组规则，虽然给不出完美解答却能找出一种解答。查找一些棋类对弈程序中用于计算下一步的启发式方法来研究一下。

14.20 另一种处理难解型问题的方法是采用满意算法（satisficing algorithm）。这一类算法能找到相当好的解，但未必是最优解。找一个能在合理时间内解决旅行商问题，但未必是最优解的算法。

深入学习

要学习更多让程序有效工作的知识，我们推荐《Structure and Interpretation of Computer Programs》[2]（中译本《计算机程序的构造和解释》，机械工业出版社。——译者注）。它没有讲时钟频率和高速缓存，而是告诉你很多如何考虑程序使之更有效的知识。

函数式编程

本章学习目标

- 使用更多函数让编程更容易。
- 使用函数式编程快速构建更强大的程序。
- 理解哪些因素使函数式编程不同于过程式编程或命令式编程。
- 学会在Python中使用else。
- 学会在Python中使用global。

15.1 使用函数简化编程

为什么要使用函数？如何使用才能简化编程？在本书中，我们多次讨论过使用多个函数编程的话题。现在，在开始以一种更强大的方式使用函数之前，我们先来总结一下它的好处。

函数能管理复杂性。可否把程序的所有代码全部写进一个大函数中呢？当然可以，只是那样一来程序就难以维护了。当程序变得越来越大，复杂度也会越来越高。我们可以使用函数：

- 隐藏细节，从而关注于自己关心的事情。
- 需要时找到修改程序的正确位置——找到有问题的函数比在几千行代码中寻找某一行容易多了。
- 使程序更易于测试和调试。

如果把程序分解成更小的片段，那么我们就可以单独测试每一小段。想想我们的HTML程序，我们可以在命令区分别测试doctype()、title()和body()这样的函数，而不必每次都测整个程序。于是，你可以单独编写更小的函数，处理更小的问题，直到所有问题解决——然后你可以忽略它们，转而关注更大的问题。

```
>>> print doctype()
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
Transition//EN" "http://www.w3.org/TR/html4/
loose.dtd">
>>> print title("My title string")
<html><head><title>My title string</title></head>
>>> print body("<h1>My heading</h1><p>My
paragraph</p>")
<body><h1>My heading</h1><p>My paragraph</p></body>
</html>
```

在查找问题（bug）的时候，能够测试上面这样的小函数是很有帮助的，它能使你信任自己的函数。你应该以各种不同方式把函数调用一下试试，确信它总能完成你所要求的任务。一旦这种信任建立起来，那么你就不用再考虑这个函数了，放心让它去完成自己的任务——你自己则去做一些更强大的事情（见下一节）。

只要选择恰当，多用一些函数可以简化对整个程序的理解。如果使用一些子函数分别完成整体功能的各个细小部分，那么我们就说函数的粒度（granularity）改变了。如果粒度太小，

那只会将一种复杂性转换成另一种复杂性。但如果粒度适当, 这些子函数就能使整个程序更易于理解和修改。

考虑下面的主页程序, 子函数的粒度比我们之前的程序小得多:



程序137: 子函数粒度更小的主页产生器

```
def makeHomePage(name, interest):
    file=open("homepage.html","wt")
    file.write(doctype())
    file.write(startHTML())
    file.write(startHead())
    file.write(title(name+"'s Home Page"))
    file.write(endHead())
    file.write(startBody())
    file.write(heading(1,"Welcome to "+ name +"'s Home Page"))
    myParagraph = paragraph( "Hi! I am " + name + ". This is my home-
page! I am interested in " + interest + "</p>" )
    file.write(myParagraph)
    file.write(endBody())
    file.write(endHTML())
    file.close()

def doctype():
    return '<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01
    Transition//EN" "http://www.w3.org/TR/html4/loose.dtd">'

def startHTML():
    return '<html>'

-这几行程序应该与下面的一行连续。Python中一条命令不可以跨多行。

def startHead():
    return '<head>'

def endHead():
    return '</head>'

def heading(level,string):
    return "<h"+str(level)+">"+string+"</h"+str(level)+">"

def startBody():
    return "<body>"

def paragraph(string):
    return "<p>"+string+"</p>"

def title(titlestring):
    return "<title>"+titlestring+"</title>"

def endBody():
    return "</body>"

def endHTML():
    return "</html>"
```

这个版本更容易测试, 但现在新的复杂性出现了: 记住刚刚取的这一大堆函数名不是件容易的事。



实践技巧：在难度大的部分使用子函数

编写程序时一旦碰到难度较大的部分，就应该把这一部分分解成子函数。这样，你可以一个函数一个函数地单独调试或修正它们。

考虑一下产生目录索引页面的程序。循环是程序中难度较大的部分，于是我们把它分解成单独的子函数，这样，要修改格式化链接的方式会更加方便——因为相关代码全部放到后面的子函数中了。



程序138：使用子函数的索引页面产生器

```
def makeSamplePage(directory):
    samplesFile=open(directory+"//samples.html","wt")
    samplesFile.write(doctype())
    samplesFile.write(title("Samples from "+directory))
    # 现在我们来组装页面主体字符串
    samples=""<h1>Samples from "+directory+" </h1>"
    for file in os.listdir(directory):
        if file.endswith(".jpg"):
            samples = samples + fileEntry(file)
    samplesFile.write(body(samples))
    samplesFile.close()

def fileEntry(file):
    samples=""<p>Filename: "+file+"<br />"
    samples=samples+'</p>'
    return samples
```

这种分解程序的方法属于过程式抽象。过程式抽象是：

- 陈述问题，弄清自己要做的事情。
- 将问题分解为一个个子问题。
- 不断将子问题分解为更小的问题，直到自己清楚如何编程解决这个更小的问题。
- 目标是让主函数明确告诉子函数要做什么。每个子函数应当完成且只完成一件逻辑任务。

可以把过程式抽象看做填充一棵函数树（如图15.1所示）。这样，修改程序就只是修改树中一个结点（函数）的问题，而添加代码也只是添加结点的问题。比如，按这种分解方式，为索引页面程序添加处理WAV文件的功能只需要修改fileEntry函数，结果就如图15.2所示。

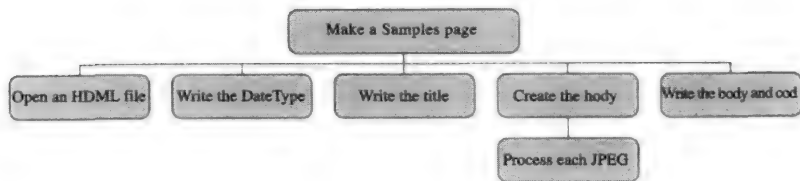


图15.1 创建索引页面的函数层次

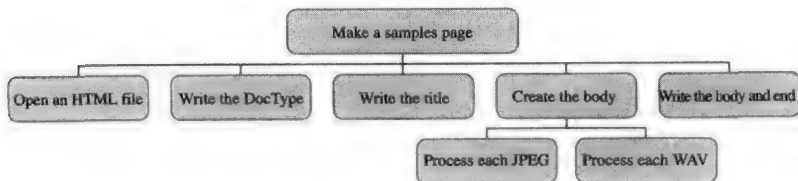


图15.2 改变程序只需对函数层次稍做调整

15.2 使用Map和Reduce进行函数式编程

如果你信任自己的函数，就能以更少的代码编出同样功能的程序。当你真正理解一个函数，并相信它完成了你想做的事情时，你就可以用短短数行代码完成令人惊叹的功能。我们可以编写将其他函数应用于数据的函数，我们甚至可以通过一种称为递归（recursion）的过程让函数调用自身。

函数同样是与值关联的名字，只不过这里的“值”是一段代码，而不是列表、数字序列或字符串。调用函数时，先给出函数的名字，然后跟上括号括起来的参数。不加括号的话，函数名仍然对应一个值——此时它对应函数的代码。函数也可以是数据——它们可作为其他函数的输入来传递。

```
>>> print makeSamplePage
<function makeSamplePage at 4222078>
>>> print fileEntry
<function fileEntry at 10206598>
```

下面的`apply()`函数接受另一个函数以及那个函数的输入（以列表的形式）作为输入。也就是说，`apply()`将函数应用到它的输入上了。

```
def hello(someone):
    print "Hello,", someone
>>> hello("Mark")
Hello, Mark
>>> apply(hello, ["Mark"])
Hello, Mark
>>> apply(hello, ["Betty"])
Hello, Betty
```

更有用的一个接受函数作为输入的函数是`map()`。`map()`函数接受一个函数和一个序列作为输入。它将输入函数应用到输入序列中的每个元素上，然后将输入函数每次返回的值汇总起来作为自己的返回值。

```
>>> map(hello, ["Mark", "Betty", "Matthew", "Jenny"])
Hello, Mark
Hello, Betty
Hello, Matthew
Hello, Jenny
[None, None, None, None]
```

`filter()`函数也接受一个函数和一个序列作为输入。它将输入函数应用到序列中的每个元素上，如果函数应用到某个元素时返回值为真（1），那么`filter`便返回那个元素；如果返回值为假（0），则跳过那个元素。我们可以用`filter()`快速抽取自己感兴趣的数据。

```
def rInName(someName):
    # 找不到时，find()会返回-1
    if someName.find("r") == -1:
        return 0
    # 如果不是-1，就说明找到了
    if someName.find("r") != -1:
        return 1
```

```
>>> rInName("January")
1
>>> rInName("July")
0
>>> filter(rInName, ["Mark", "Betty", "Matthew", "Jenny"])
['Mark']
```

上面的`rInName()`（一个当输入单词中包含“r”时，返回真的函数）可以改写成更加简短的形式。实际上，表达式本身就可以求值为1或0（真或假）。我们可以针对这些逻辑值执行某种运算。`not`就是这类逻辑运算符之一。它返回输入值的逻辑非。下面就是用逻辑运算符改过的`rInName()`。

```
def rInName2(someName):
    return not(someName.find("r") == -1)

>>> filter(rInName2, ["Mark", "Betty", "Matthew", "Jenny"])
['Mark']
```

`reduce`同样接受一个函数和一个序列，但它会把结果合并起来。下面一个例子对所有的数字求和：首先算 $1 + 2$ ，然后算 $(1 + 2) + 3$ ，然后 $(1 + 2 + 3) + 4$ ，最后算 $(1 + 2 + 3 + 4) + 5$ 。所有的加数都作为输入传入。

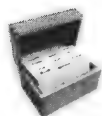
```
def add(a,b):
    return a+b

>>> reduce(add, [1,2,3,4,5])
15
```

让我们再来看看这个例子：你不觉得创建那个`add`函数有点儿浪费吗？那么小，也没做什么事。事实上，只为了使用一下的话，就不必为函数取个名字。无名的函数称为`lambda`。这是计算机科学中一个很老的术语，它的历史可以追溯到最原始的编程语言之一，Lisp。凡是可以使用函数名的地方，都可以放入一个`lambda`。`lambda`的语法是：关键字`lambda`，后面跟逗号分隔的输入变量，然后一个冒号，然后是函数体。下面是一些例子，包括用`lambda`重写的`reduce`例子。可以看到，只要给`lambda`赋个名字就可以定义函数，效果与程序区输入的那种函数完全一样。

```
>>> reduce(lambda a,b:a+b, [1,2,3,4,5])
15
>>> (lambda a:"Hello, "+a)("Mark")
'Hello, Mark'
>>> f=lambda a:"Hello, "+a
>>> f
<function <lambda> 6>
>>> f("Mark")
'Hello, Mark'
```

使用`reduce`和`lambda`，可以完成真正的计算。下面的函数计算输入参数的阶乘。数字 n 的阶乘就是小于或等于 n 的所有正整数的乘积。例如，4的阶乘就是 $(4 \times 3 \times 2 \times 1)$ 。



程序139：使用lambda和reduce计算阶乘

```
def factorial(a):
    return reduce(lambda a,b:a*b, range(1,a+1))
```

程序原理

这个程序从右往左读比较容易。首先，用`range(1, a + 1)`创建1到a之间的所有数字组成的列表。然后，用`reduce`来应用一个函数（`lambda`），从输入列表中的第一个数字开始，乘以下一个，再乘以下一个……直到全部乘完。

```
>>> factorial(2)
2
>>> factorial(3)
6
>>> factorial(4)
24
>>> factorial(10)
3628800
```

现在你可能会想：“OK，`map`、`filter`和`reduce`看起来似乎有用。或许有些时候真的有用吧。但世上会有人使用`apply`吗？它与我们手动输入函数调用完全一样，不是吗？”这没错，但实际上我们可以用`apply`实现`map`、`filter`或`reduce`。有了`apply`，当我们想手动实现`map`、`filter`或`reduce`时，就可以写出来。

```
def myMap(function, list):
    for i in list:
        apply(function, [i])

>>> myMap(hello, ["Fred", "Barney", "Wilma", "Betty"])
Hello, Fred
Hello, Barney
Hello, Wilma
Hello, Betty
```

这种编程风格称为函数式编程（functional programming）。在这之前，我们用Python做的事情可以称为过程式编程（procedural programming），因为我们关注的是过程的定义；或者叫命令式编程（imperative programming），因为我们大部分时间都在命令计算机执行动作或改写变量值（或者说改变状态）。函数作为数据，或者函数作为其他函数的输入，是函数式编程的关键思想。

函数式编程无比强大。函数可以一层层地应用于其他函数之上，最终只要短短数行代码就能完成大量动作。函数式编程常用于构建人工智能系统，也用于构建原型系统。在某些领域中，问题很难，定义也不甚清晰，于是就希望用少量代码完成大量动作——即使那几行代码多数人都难以读懂。

15.3 针对媒体的函数式编程

还记得把Katie头发调成红色的函数吗？程序36，也就是下面这个程序：

```
def turnRed():
    brown = makeColor(42,25,15)
    file="C:/ip-book/mediasources/katieFancy.jpg"
    picture=makePicture(file)
    for px in getPixels(picture):
        color = getColor(px)
        if distance(color,brown)<50.0:
            redness=int(getRed(px)*2)
            blueness=getBlue(px)
            greenness=getGreen(px)
            setColor(px,makeColor(redness,blueness,greenness))
    show(picture)
    return(picture)
```

我们可以用一行代码写出这个程序。首先，我们需要两个功能函数——一个函数检查单个像素，判断它是否应该调成红色；另一个执行实际操作，把像素调红。我们的那一行代码把满足调色规则的像素过滤（filter）出来，然后将调色函数映射（map）到这些像素上。在函数式编程中，你不必编写使用大块循环的函数，相反，只需要写个小函数并将它们应用到数据上。恰似我们把函数带给了数据，而不是让函数去获取所有的数据。



程序140：函数式的秀发染红

```
def turnHairRed(pic):
    map(turnRed, filter(checkPixel, getPixels(pic)))

def checkPixel(aPixel):
    brown = makeColor(42,25,15)
    return distance(getColor(aPixel), brown) < 50.0

def turnRed(aPixel):
    setRed(aPixel, getRed(aPixel)*2)
```

程序原理

turnRed函数接受单个像素作为输入并将它的红色数量加倍。checkPixel函数根据输入像素与棕色是否足够接近来返回真或假。turnHairRed函数接受一幅图片作为输入，并用filter将checkPixel应用于输入图片中的所有像素上（使用getPixels获得）。如果像素与棕色足够接近，那么filter就会返回它。然后，使用map将turnRed应用到filter返回的所有像素上。

以下是我们使用这个程序的方法：

```
>>> pic=makePicture(getMediaPath("KatieFancy.jpg"))
>>> map(turnRed, filter(checkPixel, getPixels(pic)))
```

不改变状态的媒体操作

函数式编程的另一个重要方面是无状态编程。怎样算改变状态呢？举例来说，颜色处理函数会改变作为输入传入的对象。良好的函数式程序不会做那样的事情。如果对象需要改变，良好的函数式程序会把对象复制一份，然后修改并返回对象副本。无状态的优势是：可以把函数嵌套起来，把一个函数的输出作为另一个函数的输入，就像嵌套调用数学函数那样，没有人期望sine(cosine(x))改变x的程度会超过sine(x)。函数式编程风格中的函数也是一样。我们说这样的函数没有“副作用”。函数只做它该做的事并返回一个结果。它们丝毫不会改动输入数据。

我们来看一下，如果把媒体操控函数写成不修改状态的风格会是什么样子。



程序141：在不修改图片的前提下修改颜色

```
def decreaseRed(aPicture):
    returnPic = makeEmptyPicture(getWidth(aPicture),getHeight(aPicture))
    for x in range(getWidth(aPicture)):
        for y in range(getHeight(aPicture)):
            srcPixel = getPixelAt(aPicture,x,y)
            returnPixel = getPixelAt(returnPic,x,y)
            setColor(returnPixel,getColor(srcPixel))
            setRed(returnPixel, 0.8*getRed(srcPixel))
    return returnPic

def increaseBlue(aPicture):
    returnPic = makeEmptyPicture(getWidth(aPicture),getHeight(aPicture))
    for x in range(getWidth(aPicture)):
        for y in range(getHeight(aPicture)):
            srcPixel = getPixelAt(aPicture,x,y)
            returnPixel = getPixelAt(returnPic,x,y)
            setColor(returnPixel,getColor(srcPixel))
            setBlue(returnPixel, 1.2*getBlue(srcPixel))
    return returnPic
```

程序原理

两个程序的基本结构是一样的。函数中创建了与输入图片尺寸相同的待返回对象：returnPic。把输入图片中每个像素的颜色都复制到returnPic图片的相应像素中。然后，减少图片中的红色，或者增加蓝色，最后返回returnPic。

有了这些函数，我们就可以在图片中应用这类功能，且不会改变原来的图片。我们可以把函数随意嵌套使用。现在，这些函数更像数学函数了。

```
>>> newp = increaseBlue(decreaseRed(p))
>>> show(newp)
>>> show(decreaseRed(p))
>>> show(decreaseRed(increaseBlue(p)))
>>> show(increaseBlue(p))
```

15.4 递归：一种强大的思想

递归就是编写调用自身的函数。在递归中，并不直接使用循环，而是编写一个函数不断调用它自己，从而自动循环起来。编写递归函数至少要包含两样东西：

- 结束时要做的事（比如处理到最后一个数据项时）；
- 数据较多时要做的事，通常是处理一个数据元素，然后再次调用函数自身来处理剩下的元素。

在使用递归处理媒体之前，我们先用一个简单的文本函数考查一下这种机制。我们考虑如何编写一个输出如下结果的函数。

```
>>> downUp("Hello")
Hello
ello
```

```
llo
lo
o
lo
llo
ello
Hello
```

递归理解起来会让人纠结，要理解它需仰仗你对函数的信任。函数是否做了它该做的事？如果是就调用它——它会把事情做对的。

为帮助你理解递归，我们将以三种方式来讨论它。第一种是**过程式抽象**——不断分解问题，直到分解成可以轻松编写函数的最小片段，然后尽可能地重用编写出的函数。

首先，针对只含一个字符的单词考虑downUp。这个简单：

```
def downUp1(word):
    print word

>>> downUp1("I")
I
```

现在，针对双字符单词做downUp。我们将重用downUp1，因为它已经有了。

```
def downUp2(word):
    print word
    downUp1(word[1:])
    print word

>>> downUp2("it")
it
t
it
>>> downUp2("me")
me
e
me
```

现在考虑三字符单词：

```
def downUp3(word):
    print word
    downUp2(word[1:])
    print word

>>> downUp3("pop")
pop
op
p
op
pop
>>> downUp3("top")
top
op
p
op
top
```

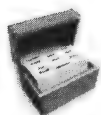
有没有看出一种模式来？让我们试试：

```
def downUpTest(word):
    print word
    downUpTest(word[1:])
    print word
>>> downUpTest("hello")
hello
ello
llo
lo
o
The error was:java.lang.StackOverflowError
I wasn't able to do what you wanted.
The error java.lang.StackOverflowError has occurred
Please check line 101 of C:\ip-book\programs\
functional
```

常规Python中给出的错误信息可能稍有不同，但基本意思是一样的。

```
>>> downUpTest("hello")
...
File "<stdin>", line 3, in downUpTest
File "<stdin>", line 3, in downUpTest
RuntimeError: maximum recursion depth exceeded
```

发生什么事了？处理到只剩一个字符之后，程序不断调用downUpTest，直到耗尽某一段称为栈（stack）的内存区域。我们需要一种方法来告诉函数“如果只剩下一个字符，就把它打印出来，然后别再调用自己了！”下面的函数可以正常工作。



程序142：递归的downUp

```
def downUp(word):
    print word
    if len(word)==1:
        return
    downUp(word[1:])
    print word
```

第二种考虑递归的方式基于我们的陈年老友：跟踪。在下面的描述中，缩进的部分是注解。

```
>>> downUp("Hello")
```

	len(word)不等于1，于是我们打印这个单词
Hello	
	此时调用downUp("ello") 字符仍然不止一个，仍打印单词
ello	
	此时调用downUp("llo") 字符仍然不止一个，仍打印单词
llo	
	此时调用downUp("lo") 字符仍然不止一个，仍打印单词
lo	
	此时调用downUp("o") 字符仍然不止一个，仍打印单词
o	

(续)

lo	此时调用downUp("o") 只剩一个字符了！打印它，然后返回
o	此时downUp("lo")从downUp("o")之后继续执行 再次打印并结束
lo	此时downUp("llo")继续执行（从downUp("lo")返回之后） 打印并结束
llo	此时downUp("ello")继续执行 打印并结束
ello	最后，原先downUp("Hello")的最后一行可以执行了
Hello	

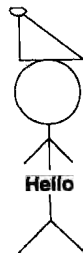
第三种考虑递归的方式是把函数调用想象成小精灵——计算机中的一个小人，你让他做什么，他就做什么。

以下是downUp发给小精灵的指令：

- 1) 接受一个单词作为输入。
- 2) 如果单词中只有一个字符，那么就把它写到屏幕上，然后你的任务完成，可以停下来坐坐了。
- 3) 把单词写到屏幕上。
- 4) 雇用另一个小精灵来执行同样一套指令，把你的单词去掉第一个字符后传给新的小精灵。
- 5) 等待你雇用的小精灵完成任务。
- 6) 再次把你的单词写到屏幕上。你的任务完成。

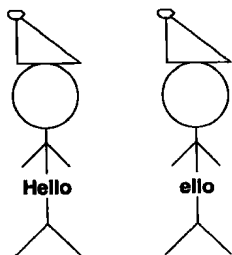
建议大家在课堂上尝试一下这个过程——会很有趣，而且能帮助大家理解递归。整个过程就像下面这样：

- 首先用“Hello”作输入雇用第一个小精灵。

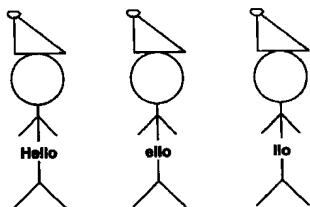


(可以把这个小人看做小精灵的抽象)

- 携带“Hello”的小精灵开始执行指令。他接受单词作为输入，发现自己手里不只一个字符，于是把单词Hello写到屏幕上。然后，他雇用一个新的小精灵，并将“ello”作为她的输入。



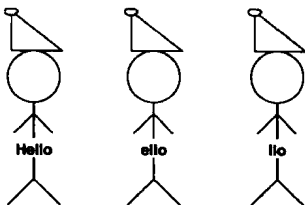
- 携带“ello”的小精灵接受了输入，发现自己手里不只一个字符，于是把单词写到屏幕上（紧跟在Hello下面输出ello）。然后她雇用了新的小精灵，并向他输入了“llo”。



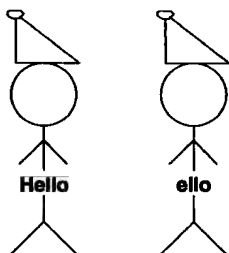
- 这时我们可以看出一些规律。每个小精灵只知道他后面一个小精灵——也就是他雇用的那个。他必须等那个小精灵结束，然后自己才结束。工作进入收尾阶段后，首先结束的是右边的小精灵，也就是说，后雇用的先结束。这样的结构我们称之为栈（stack）——小精灵们自左向右排列成栈，最后列入的将是最先退出的。

如果最早输入的是个很长的单词（比如“antidisestablishmentarianism”），可以想象，可能没有足够的空间让所有小精灵排列成栈，我们把这一结果称为栈溢出（stack overflow）——如果递归得太深（也就是小精灵太多），Python就会报告栈溢出错误。

- 想象我们继续这一模拟过程，先后为“lo”和“o”雇用了小精灵。“o”的那个写下她的o以后便坐下休息了。
- 携带“lo”的那个小精灵现在准备结束。她把lo写到屏幕上——在o的下面，而o在上次的lo下面。这个小精灵也坐下了。这样，我们还有3个小精灵在等待结束。



- 携带“llo”的那个小精灵写下llo，然后坐下。



- 携带“ello”的小精灵一直在等待“llo”的那个结束。现在她写下ello然后坐下。



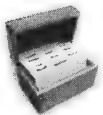
- 最后携带“Hello”的小精灵二次写下Hello，然后也坐下。此时栈空了。

为什么要使用函数式编程和递归呢？因为借助于它们，你可以用很短的代码完成大量的工作。处理困难问题时，它们是非常有用的技术。任何循环都可以用递归实现。为此，许多人都觉得递归是最灵活、最优雅、最强大的循环形式。

15.4.1 递归式目录遍历

在本书中，我们最早遇到的递归结构是目录树。文件夹可以包含其他文件夹，这样一层层包含下去，可以有任意多层。遍历目录结构（访问到每个文件）最简单的方法就是使用递归。

我们已经知道如何获得一层目录中的所有文件——使用`os.listdir`。问题在于，我们需要确定其中哪些是目录。幸运的是，Java知道如何判断——使用它的File对象。于是，我们也可以方便地从Jython中使用它。如果发现某一项是目录时，我们就可以像处理第一层目录那样处理它。



程序143：打印目录树中的所有文件名

```
import os
import java.io.File as File

def printAllFiles(directory):
    files = os.listdir(directory)
    for file in files:
        fullname = directory+"/"+file
        if isDirectory(fullname):
            printAllFiles(fullname)
        else:
            print fullname

def isDirectory(filename):
    filestatus = File(filename)
    return filestatus.isDirectory()
```

程序原理

我们需要用Python的`os`模块和Java的`java.io.File`类。为了打印给定目录中的所有文件，使用`os.listdir`取得目录中的所有文件组成的列表。针对列表中的每一项，把目录与文件名连在一起，获得一个完整路径。然后，通过`isDirectory`询问该路径是否为目录。在`isDirectory`函数中，使用Java的File对象来询问一个路径是否为目录并返回结果。

在本书中，每次需要测试一个条件时，我们常常使用两条if语句。因此，上面的函数也可

以这么写：

```
def printAllFiles(directory):
    files = os.listdir(directory)
    for file in files:
        fullname = directory+"/"+file
        if isDirectory(fullname):
            printAllFiles(fullname)
        if not isDirectory(fullname):
            print fullname
```

然而，每次执行isDirectory测试都会引发相当复杂的文件操作，对执行时间来说，这种做法代价太高了。所以，最好不要重复使用它，我们可以说：“如果不是目录，也就是‘else’，就这样……”于是我们便使用了else这一设施。在可读性方面else可能不如两次if那么好，但它能防止重复测试一件事情，因此效率更高。

我们将使用图15.3所示的文件夹来测试这一函数。

```
>>> printAllFiles("/home/guzdial/Documents/sampleFolder")
/home/guzdial/Documents/sampleFolder/
  blueMotorcycle.jpg
/home/guzdial/Documents/sampleFolder/sounds/
  bassoon-c4.wav
/home/guzdial/Documents/sampleFolder/sounds/
  bassoon-g4.wav
/home/guzdial/Documents/sampleFolder/sounds/
  bassoon-e4.wav
/home/guzdial/Documents/sampleFolder/birds/bird3.jpg
/home/guzdial/Documents/sampleFolder/birds/bird2.jpg
/home/guzdial/Documents/sampleFolder/birds/bird1.jpg
/home/guzdial/Documents/sampleFolder/birds/bird5.jpg
/home/guzdial/Documents/sampleFolder/birds/bird4.jpg
/home/guzdial/Documents/sampleFolder/birds/bird6.jpg
/home/guzdial/Documents/sampleFolder/blue-mark.jpg
/home/guzdial/Documents/sampleFolder/butterfly.jpg
```

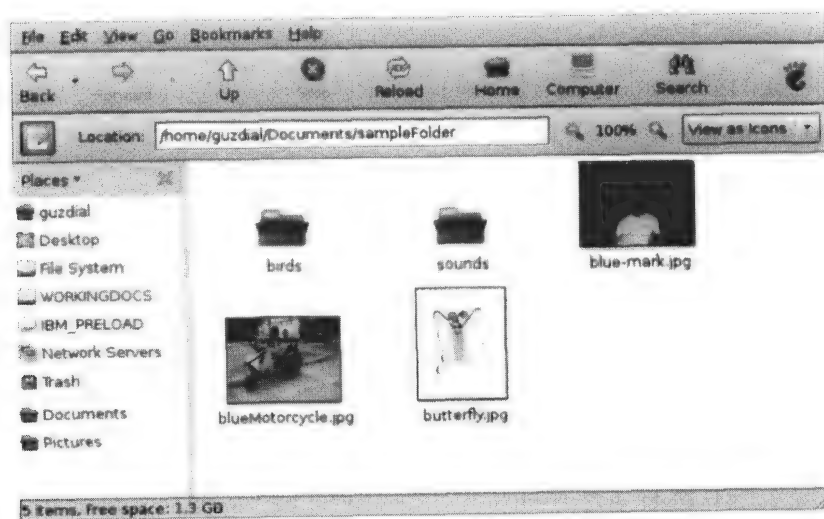


图15.3 sampleFolder的内容

15.4.2 递归式媒体函数

我们可以考虑用递归方式编写decreaseRed()这样的媒体函数，就像下面这样：



程序144：用递归方法减少红色

```
def decreaseRedR(aList):
    if aList == []: # 空列表
        return
    setRed(aList[0], getRed(aList[0]) * 0.8)
    decreaseRedR(aList[1:])
```

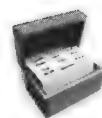
程序原理

若输入像素列表为空，则程序便停止（return）；否则，我们取得列表中的第一个像素（aList[0]）并将它的红色值降低20%（乘以0.8）。然后，对列表中的其他像素（aList[1:]）调用decreaseRedR。

用命令decreaseRedR(getPixels(pic))来调用这一函数。警告：即使处理中等大小的图片，程序也可能没法正常工作。Python（以及Jython底下的Java）可没想到会处理这么深的递归，于是内存用光了。如果图片非常小，程序倒是可以工作。这个decreaseRed版本实际上有两个问题：

- 首先，它针对每个像素递归一次。搞不好就要递归数10万次。
- 其次，每次递归调用，它都传递一份像素列表。这意味着每处理一个像素，内存中都要保存一份像素列表的副本。如此的内存占用量可太大了。

我们可以换种方法编写这个函数，从而纠正第二个问题。只要把保存像素列表的变量声明为global，就可以避免每次都传递完整的像素列表。那样一来，多个函数就可以共享这个变量。



程序145：使用全局变量的递归decreaseRed

```
aPicturePixels = []

def decreaseRedR(aPicture):
    global aPicturePixels
    aPicturePixels = getPixels(aPicture)
    decreaseRedByIndex(len(aPicturePixels) - 1)

def decreaseRedByIndex(index):
    global aPicturePixels
    pixel = aPicturePixels[index]
    setRed(pixel, 0.8 * getRed(pixel))
    if index == 0: # 空了
        return
    decreaseRedByIndex(index - 1)
```

程序原理

首先，我们在所有函数之外创建了aPicturePixels。global语句告诉Python：aPicturePixels变量应该定义在文件（模块）范围内，而不是在函数局部。decreaseRedR把所有的像素放入aPicturePixels，然后使用像素列表的最后一个下标，即列表的长度（len）

减1，调用辅助函数decreaseRedByIndex。decreaseRedByIndex从指定位置取得像素并将其中的红色数量降至原来的80%。如果下标为0，也就是处理到第一个像素的时候，便从函数中返回，从而结束处理；如果不是0，就把当前下标减1，接着降低它前面一个像素的红色值。

问题在于，这个版本仍然针对每个像素递归一层。即使图片不大（比如640×480），在处理完整幅图片之前还是会栈溢出。一般来讲，在Jython中逐像素做递归处理十分困难，可能根本做不到。

编程摘要

以下是我们在这章见过的一些函数或程序片段：

函数式编程

apply	接受一个函数和一个列表作为输入，其中列表中的元素个数与输入函数接受的参数个数一致，然后apply基于列表中的输入参数调用传入的函数
map	接受一个函数和一个含有多项输入的列表作为输入。针对每个列表元素调用传入的函数，并返回各输出结果（返回值）组成的列表
filter	接受一个函数和一个含有多项输入的列表作为输入。针对每个列表元素调用传入的函数，如果输入函数返回真（非0），则返回的结果中将包含该元素
reduce	接受一个函数和一个含有多项输入的列表作为输入。输入函数首先应用到前两个列表元素上，返回的结果与下一个列表元素一起作为下一次函数调用的输入，返回的结果再跟下一个列表元素一起作为再下一次函数调用的输入，依此类推。最后一次调用的结果作为reduce的结果返回
else:	位于if之后，仅当if语句的测试结果为假时执行else:之后的代码块
global	global语句中列出的变量引用的是函数之前在文件（或模块）范围内创建的版本。这样我们可以共享对象的引用，而不是复制它

习题

- 15.1 这是一道智力题。有6个方块，其中一个比其他5个重。有一架天平，但只能使用两次。请找出最重的一个方块。（a）把你设计的过程编写成算法。（b）这个算法类似于哪种查找？
- 15.2 编写函数计算一个数的斐波纳契（Fibonacci）数。0的斐波纳契数为0，1的斐波纳契数为1。 n 的斐波纳契数 $Fib(n) = Fib(n - 2) + Fib(n - 1)$ 。
- 15.3 编写函数将摄氏温度转化为华氏温度。
- 15.4 基于一个人的体重和身高，编写函数计算他的身体质量指数（body mass index）。
- 15.5 编写函数按20%的比例计算小费。
- 15.6 查一查什么是谢尔宾斯基三角（Sierpinski's triangle）。编写递归函数创建一个谢尔宾斯基三角。
- 15.7 查一下什么是科赫雪花（Koch's snowflake）。编写递归函数创建科赫雪花。
- 15.8 把程序140中的功能函数改写为lambda，从而把turnHairRed()函数改成只有一行。
- 15.9 描述以下函数的功能，并尝试一下不同的数字输入。

```
def test(num):
    if num > 0:
        return test(num-1)
```

```

    else:
        return 0

```

15.10 描述以下函数的功能，并尝试一下不同的数字输入。

```

def test(num):
    if num > 0:
        return test(num-1) + num
    else:
        return 0

```

15.11 描述以下函数的功能，并尝试一下不同的数字输入。

```

def test(num):
    if num > 0:
        return test(num-1) * num
    else:
        return 0

```

15.12 描述以下函数的功能，并尝试一下不同的数字输入。

```

def test(num):
    if num > 0:
        return num - test(num-1)
    else:
        return 0

```

15.13 描述以下函数的功能，并尝试一下不同的数字输入。

```

def test(num):
    if num > 0:
        return test(num-2) * test(num-1)
    else:
        return 0

```

15.14 编写一个递归函数，列举一个目录及其子目录中的所有文件。

15.15 编写如下的upDown()函数：

```

>>> upDown("Hello")
Hello
Hell
Hel
He
H
He
Hel
Hell
Hello

```

试着分别用递归和非递归方法实现它。哪一种更容易？为什么？

15.16 试着用filter和map这样的结构把前几章中一些声音或图片的示例程序改成函数式风格。

面向对象编程

本章学习目标

- 使用面向对象编程使程序更易于团队开发、更健壮、更易于调减。
- 理解多态、封装、继承和聚合等面向对象特性。
- 学会根据不同目标选择不同的编程风格。

16.1 对象的历史

如今，最常用的编程风格是**面向对象编程**（object-oriented programming）。我们将对照着到目前为止一直在用的过程式编程（procedural programming）风格来定义面向对象编程。

在20世纪60到70年代，过程式编程是主流的编程形式。人们使用**过程式抽象**（procedural abstraction），在不同层次上定义大量函数，并尽可能地重用这些函数。从某种程度上讲，这种方法效果相当好。然而，随着程序真正变得越来越大、越来越复杂，而且许多程序员同时开发一个程序，过程式编程就显得吃力了。

程序员们遭遇了过程冲突的问题。一个人编写的程序会以出乎他人预料的方式修改数据。也可能大家使用了同样的函数名，最终发现代码不能集成到更大的程序中。

过程式方法在考虑程序及其完成的任务方面也有问题。过程对应的是**动词**——告诉计算机做这件事，告诉计算机做那件事。但这是不是最恰当的考虑问题的方式呢？人们并不清楚。

面向对象编程是面向**名词**的编程。构造一个面向对象程序时，人们首先思考问题域中的名词——问题及其解决方案中包含哪些人和事物。确定各个对象、（对问题而言）每个对象知道什么，以及每个对象要做什么的过程称为**面向对象分析**（object-oriented analysis）。

在面向对象式的编程中，你将为对象定义变量（称为**实例变量**）和函数（称为**方法**）。你很少会有，甚至没有那种可以到处访问的全局函数或变量。相反，程序中会有一些相互通话的对象，彼此通过方法调用让对方做事情。面向对象编程的先锋之一，Adele Goldberg将这种方式称为“有事说，别碰我”（Ask, don't touch）。你不能直接“触碰”数据并为所欲为——相反，你应该通过对象的方法“请求”它处理自己的数据。

“面向对象编程”这一术语是Alan Kay发明的。Kay是一位聪明的跨学科人才——他拥有数学和生物学本科学历，又是计算机科学博士，还是一名爵士乐吉他手。2004年，他获得了ACM图灵奖，这算得上计算机领域的诺贝尔奖了。Kay把面向对象编程看做一种可以真正扩展到大型系统的软件开发方式。他还把对象描述为生物学上的细胞——按照定义明确的方式协同工作，从而使整个机体正常运转。与细胞类似，对象可以：

- 将责任分布到许多对象中，而不是聚集在一个大程序中，从而协助管理复杂性。
- 让对象相对独立地工作，以支持健壮性。
- 支持重用，因为每个对象都会向其他对象提供服务（对象可通过自己的方法为其他对象完成任务），就像真实世界中的对象一样。

从名词开始构造程序的提法也是Kay愿景的一部分。他说，软件实际是现实世界的模拟。

让软件为世界建模，构造软件的方法就会变得更清楚。你观察世界及其运作方式，然后直接复制软件中。世界上的事物知道其他事物——这便是**实例变量**的原型。世界上的事物可以做事——这便是**方法**的原型。

16.2 使用“小海龟”

20世纪60年代末，麻省理工学院（MIT）的Seymour Papert博士使用小海龟机器人帮助孩子们思考如何定义过程。小海龟的中间有一支画笔，可以抬起或放下，从而留下一条移动轨迹。后来有了图形显示设备，他便开始使用计算机显示器上的虚拟海龟，而不再使用机器人海龟。

我们会构造一些在某个“世界”里移动的小海龟对象。小海龟知道如何向前移动，如何左转，右转，或者旋转指定的角度。小海龟中间有一支画笔，可以留下移动轨迹。“世界”维护着它里边的海龟。

16.2.1 类和对象

计算机如何知道我们的“海龟”或“世界”是什么意思呢？我们必须定义海龟是什么，它知道什么，它能做什么。我们也必须定义世界是什么，它知道什么，它能做什么。在Python中，我们通过定义类来实现这种目的。类定义了该类的对象（实例）知道什么，可以做什么。我们已经创建了相应的类，这些类定义了我们所谓的“海龟”和“世界”的含义。

16.2.2 创建对象

面向对象的程序由对象构成。但我们如何创建这些对象呢？类知道该类的对象需要维护哪些信息，应该能做哪些事情，因此，应该由类来创建该类的对象。你可以将类视为对象工厂。工厂能创建很多对象。类也有点像饼干模具，你可以用模具做出很多饼干，而且它们都有同样的形状。你还可以把类看成一份设计蓝图，然后将对象看成根据蓝图建造出来的房子。

为创建并初始化一个“世界”或“海龟”对象，可以使用makeClass(parameterList)这样的语句，其中parameterList是参数列表，它包含用于初始化新对象的一系列数据项目。下面我们来创建一个“世界”对象。

```
>>> makeWorld()
```

这条命令将创建一个“世界”对象，并显示一个窗口展现这个“世界”（如图16.1所示）。开始时它只是一张全白的图片，显示在一个标题为“World”的窗口中。

我们没有给“世界”起名字，因此也无法引用它。让我们重新来过，这次将创建出来的“世界”对象命名为earth，然后在名为earth的“世界”上创建一个海龟对象。把海龟对象命名为tina。

```
>>> earth = makeWorld()
>>> tina = makeTurtle(earth)
>>> print tina
No name turtle at 320, 240 heading 0.0.
```

海龟对象出现在“世界”的中心（320，240），面朝北方（朝向heading为0）（如图16.2所示）。我们还没有为海龟（不是海龟对象）取个名字。

我们可以创建许多海龟对象。每只新的海龟都会显示在“世界”的中心。

```
>>> sue = makeTurtle(earth)
```

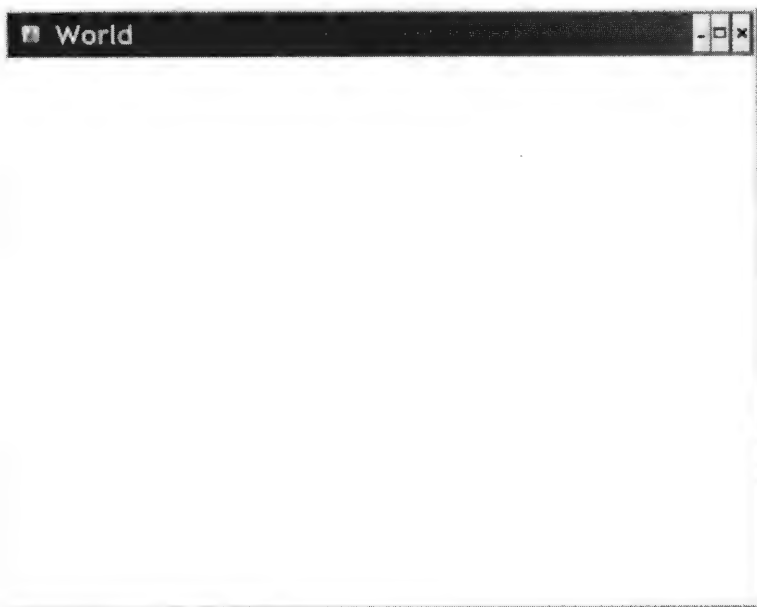


图16.1 显示一个“世界”对象

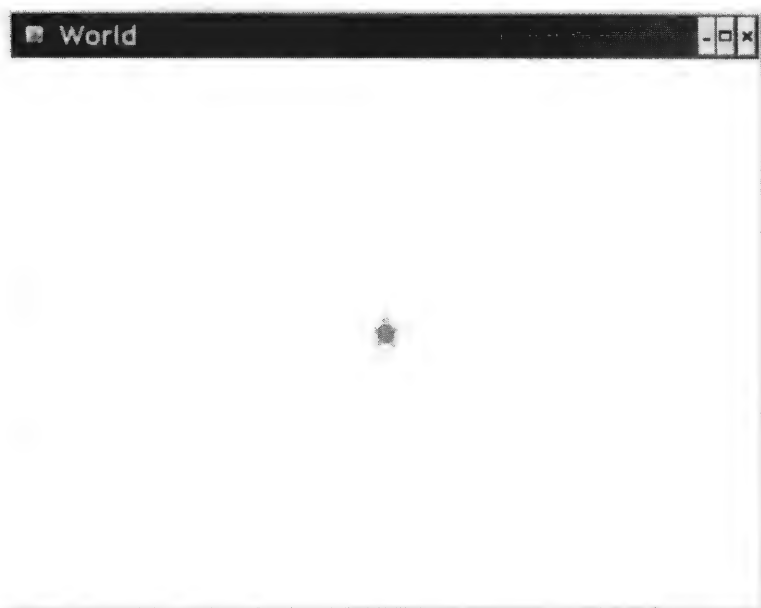


图16.2 在“世界”中创建一只小海龟

16.2.3 向对象发送消息

可以向海龟发送一条消息，以便让它做点儿事情。发送消息使用点号语法。通过点号语法，这样让对象做事：首先指出对象的名字，然后加一个点号，再加上要执行的函数名字 (`name.function(parametersList)`)。在10.3.1节我们曾见过点号语法用在字符串上的情形。

```
>>> tina.forward()
>>> tina.turnRight()
>>> tina.forward()
```

注意，只有执行动作的那只小海龟移动了（如图16.3所示）。也可以让另外一只小海龟执行动作，从而让它也移动移动。

```
>>> sue.turnLeft()
>>> sue.forward(50)
```

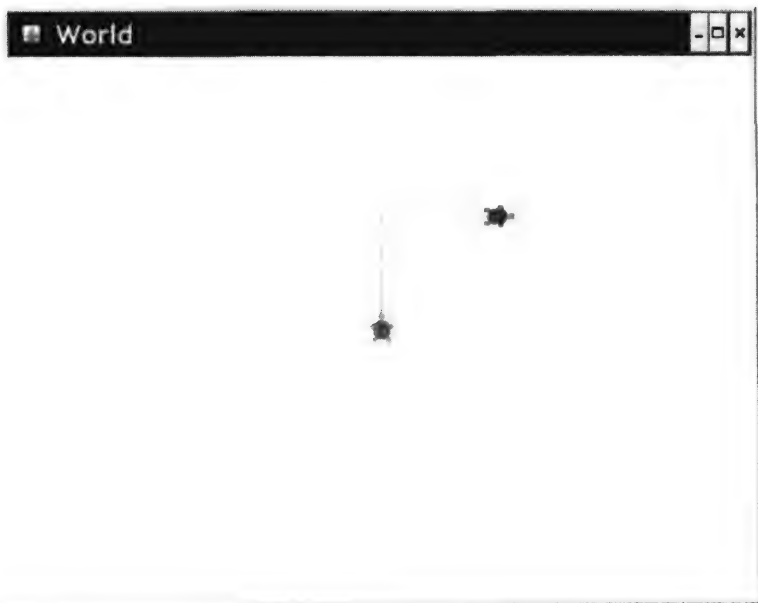


图16.3 让一只小海龟移动并转弯，另一只留在原地

注意，不同的小海龟颜色不一样（如图16.4所示）。可以看到，小海龟知道如何左转和右转：turnLeft(), turnRight()。它们还能通过forward()沿着当前的朝向前行。默认情况下，它们会前行100像素的距离。但我们也可以指定前行的像素数，比如forward(50)。海龟还知道怎样旋转指定的度数。正的度数让小海龟右转，负的度数让小海龟左转（如图16.5所示）。

```
>>> tina.turn(-45)
.>> tina.forward()
```

16.2.4 对象控制自己的状态

在面向对象编程中，通过发送消息让对象做事。对象可以拒绝你让它做的事情。什么情况下对象会拒绝呢？如果让对象做一些导致它内部数据出错的事情，那么它就应当拒绝。海龟生活的“世界”宽640像素，高480像素。如果你让海龟走出“世界”的边缘，结果会怎样呢？

```
>>> world1 = makeWorld()
>>> turtle1 = makeTurtle(world1)
>>> turtle1.forward(400)
>>> print turtle1
No name turtle at 320, 0 heading 0.0.
```

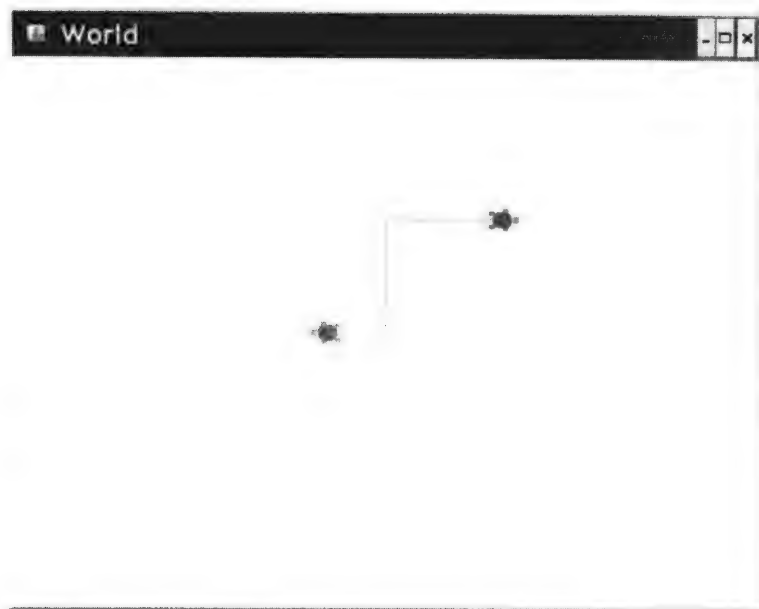
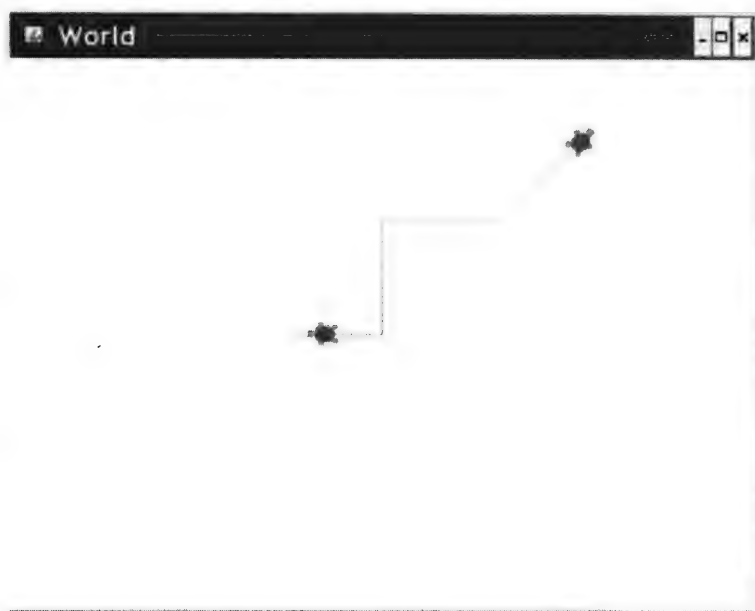


图16.4 第二只小海龟移动之后

图16.5 旋转指定的度数 (-45°)

别忘了，开始时小海龟的位置在 $(320, 240)$ ，朝向北方（上）。“世界”左上角的坐标为 $(0, 0)$ ， x 向右递增， y 向下递增。如果让海龟前行 400 像素，实际就是让它走到 $(320, 240 - 400)$ ，结果为 $(320, -160)$ 。然而，小海龟不愿意离开“世界”，当中心位于 $(320, 0)$ 时，它就停下来了（如图 16.6 所示）。这意味着任何一只小海龟都不会离开我们的视线。

16.2.5 小海龟的其他函数

除了前行和转弯外，小海龟还能做许多其他动作。可能你已经注意到了，小海龟移动的时

候会画出一道与自身颜色相同的线。可以用`penUp()`让海龟抬起画笔，用`penDown()`让小海龟放下画笔，用`moveTo(x, y)`让小海龟移到指定位置。如果让小海龟移向新位置时，画笔处于放下状态，那么小海龟就会从旧位置到新位置画一条直线（如图16.7所示）。

```
. >>> worldX = makeWorld()
>>> turtleX = makeTurtle(worldX)
>>> turtleX.penUp()
>>> turtleX.moveTo(0,0)
>>> turtleX.penDown()
>>> turtleX.moveTo(639,479)
```

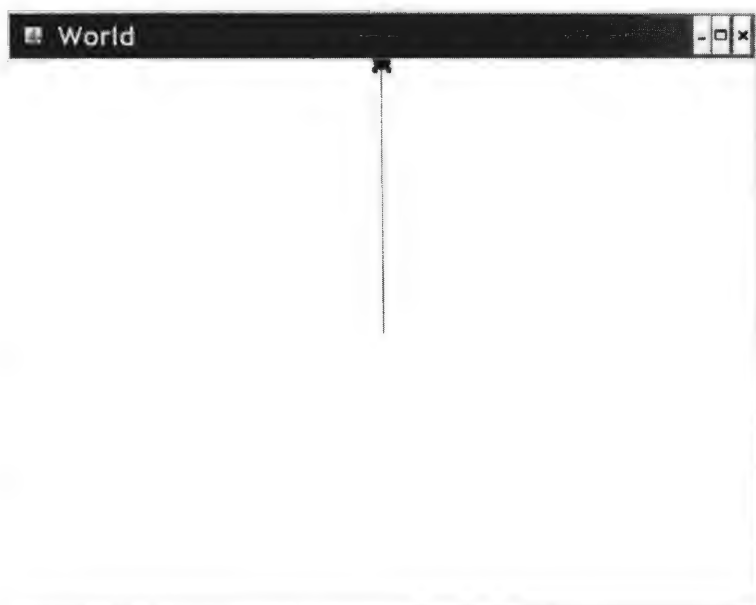


图16.6 停在“世界”边缘的小海龟

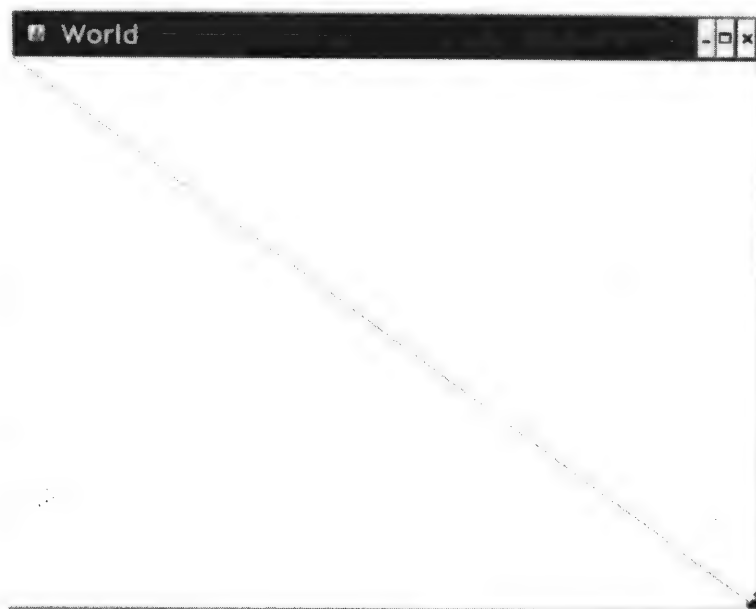


图16.7 用小海龟画对角线

可以用`setColor(color)`改变小海龟的颜色,用`setVisible(false)`让小海龟不再出现,用`setWidth(width)`改变画笔的粗细(如图16.8所示)。

```
>>> worldX = makeWorld()
>>> turtleX = makeTurtle(worldX)
>>> turtleX.setVisible(False) #don't draw the turtle
>>> turtleX.penUp() # don't draw the path
>>> turtleX.moveTo(0,240)
>>> turtleX.penDown() # draw the path
>>> turtleX.setPenWidth(100) # width of pen
>>> turtleX.setColor(blue)
>>> turtleX.turnRight()
>>> turtleX.forward(300)
>>> turtleX.penUp() # don't draw the path
>>> turtleX.setColor(red)
>>> turtleX.moveTo(400,0)
>>> turtleX.turnRight()
>>> turtleX.setPenWidth(160)
>>> turtleX.penDown() # draw the path
>>> turtleX.forward(400)
```

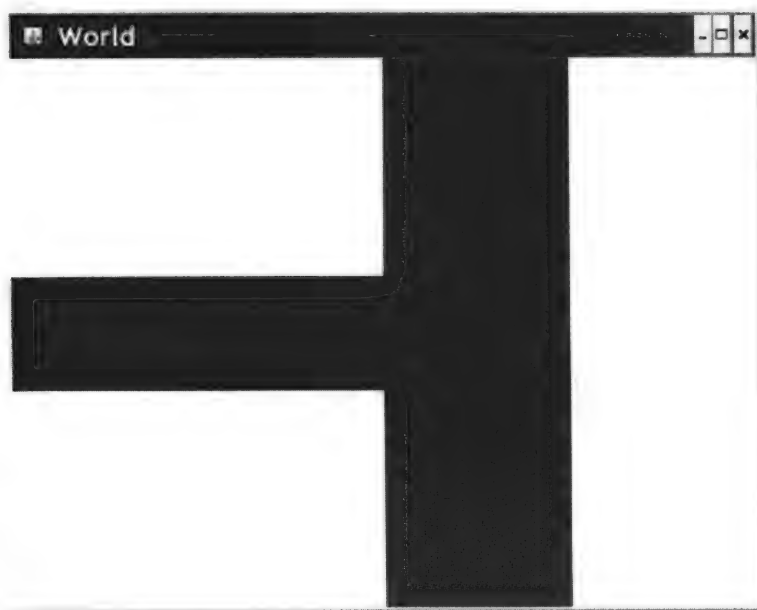
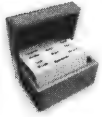


图16.8 用小海龟画矩形

16.3 教小海龟新的技艺

`Turtle`类是我们帮你定义好的。如果你想创建自己的海龟类型并教它执行新的动作,那么又该如何做呢?我们可以创建一种新的海龟类型,原先的小海龟会做的它都会做,同时又可以为它添加新的功能。这种做法称为创建子类(subclass)。正如父母眼睛的颜色会被孩子继承,原来的小海龟知道的、会做的,也统统被子类继承。子类也称为孩子类(child class),而它所继承的类称为父类(parent class)或超类(superclass)。

我们把小海龟的子类命名为SmartTurtle，为它添加一个方法，使我们的新海龟能画出正方形。方法（method）就像我们一直在使用的函数，只是它定义在类的内部，而且接受一个该类对象的引用，从哪个对象上调用方法，这个引用就指向哪个对象（通常称为self）。为画出一个正方形，新海龟会分4次右转并前行。别忘了新海龟从Turtle类那里继承了右转（turnRight）和前行（forward）的能力。



程序146：定义子类

```
class SmartTurtle(Turtle):

    def drawSquare(self):
        for i in range(0,4):
            self.turnRight()
            self.forward()
```

由于SmartTurtle是一种Turtle，使用它的方法与使用Turtle基本一致。但我们要使用新的方法来创建SmartTurtle。我们一直在使用makePicture、makeSound、makeWorld和makeTurtle这样的函数构造对象。它们都是为简化这类对象的构造而创建的。然而，用来创建对象的实际方法是使用ClassName(parameterList)这种形式。因此，要创建一个“世界”对象，可以使用：worldObj = World()；要创建SmartTurtle，可以使用：turtleObj = SmartTurtle(worldObj)。

```
>>> earth = World()
>>> smarty = SmartTurtle(earth)
smarty.drawSquare()
```

现在，我们有了一只只会画正方形的聪明海龟（如图16.9所示）。但它只能画尺寸为100的正方形。考虑到最好能画出不同尺寸的正方形，我们可以再添加一个函数，让它接受一个指定正方形边长的参数。

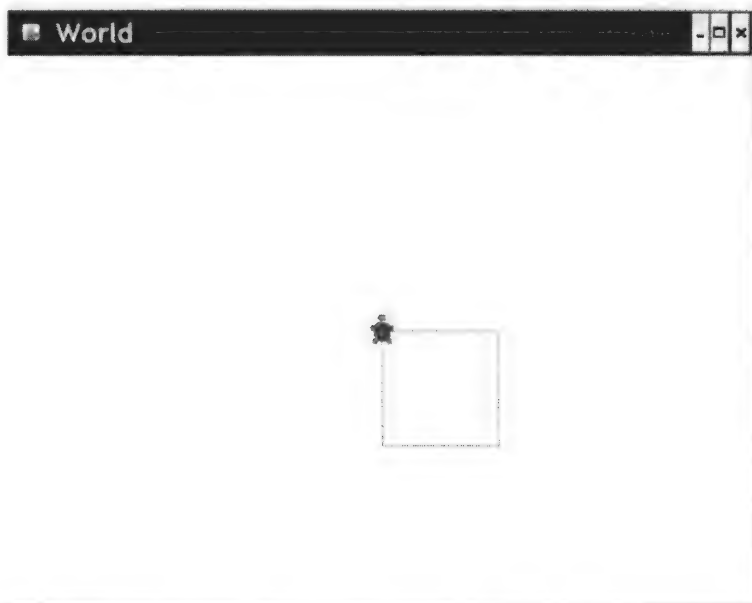


图16.9 使用SmartTurtle画正方形



程序147：定义子类

```
class SmartTurtle(Turtle):

    def drawSquare(self):
        for i in range(0,4):
            self.turnRight()
            self.forward()

    def drawSquare(self, width):
        for i in range(0,4):
            self.turnRight()
            self.forward(width)
```

你可以用这个类画出不同尺寸的正方形，如图16.10所示。

```
>>> mars = World()
>>> tina = SmartTurtle(mars)
>>> tina.drawSquare(30)
>>> tina.drawSquare(150)
>>> tina.drawSquare(100)
```

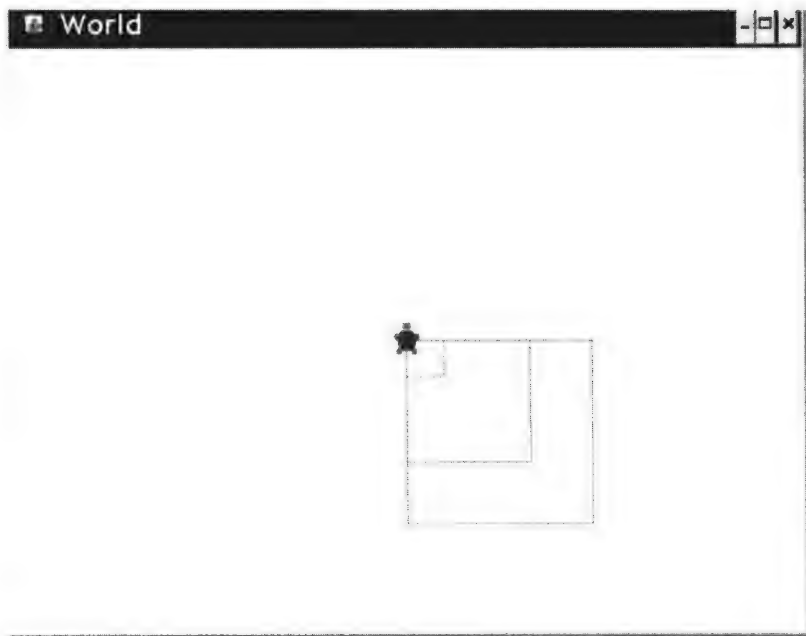


图16.10 画不同尺寸的正方形

16.4 面向对象的幻灯片

现在，我们用面向对象方法来构造一套幻灯片。假如我们想显示一幅图片，播放一段相关的声音，然后等声音播放结束再转到下一幅图片。我们将使用（前面章节提到过的）`blockingPlay()`函数，这个函数播放一段声音，然后等声音结束才执行下一条语句。



程序148：写成一个大数据的幻灯片程序

```
def playSlideShow():
    pic = makePicture(getMediaPath("barbara.jpg"))
    sound = makeSound(getMediaPath("bassoon-c4.wav"))
    show(pic)
    blockingPlay(sound)
    pic = makePicture(getMediaPath("beach.jpg"))
    sound = makeSound(getMediaPath("bassoon-e4.wav"))
    show(pic)
    blockingPlay(sound)
    pic = makePicture(getMediaPath("church.jpg"))
    sound = makeSound(getMediaPath("bassoon-g4.wav"))
    show(pic)
    blockingPlay(sound)
    pic = makePicture(getMediaPath("jungle2.jpg"))
    sound = makeSound(getMediaPath("bassoon-c4.wav"))
    show(pic)
    blockingPlay(sound)
```

从各个方面来看，这都不是个好程序。首先，从过程式编程的角度来看，这里的重复代码的多得令人发指。去掉这些重复代码程序会好一些。然后，从面向对象编程的角度来看，使用幻灯片对象会更好一些。

之前提过，对象有两大部分。首先对象知道一些东西——这些东西成了实例变量。其次对象能做一些事情——这些事情成了方法。不管是实例变量还是方法，我们都使用点号语法访问它们。

那么，幻灯片知道什么呢？它知道属于自己的图片和声音。幻灯片又能做什么呢？它能够显示自己，也就是显示它的图片，播放它的声音。

要在Python（或其他任何面向对象的编程语言，包括C++和Java）中定义一个幻灯片对象，必须定义一个幻灯片类。类为一组对象定义了实例变量和方法——也就是类的各个对象知道什么，能做什么。类的每个对象都称为类的实例（instance）。我们将构造幻灯片类的多个实例，从而制作多张幻灯片——正如我们的身体会制造很多肾脏细胞和很多心脏细胞，每个细胞都能完成特定的任务。

在Python中，创建一个类首先要写下面这样的开头：

```
class slide:
```

在这之后是缩进的方法，用于创建新的幻灯片或者显示幻灯片。我们来给slide类添加show()方法。

```
class slide:
    def show(self):
        show(self.picture)
        blockingPlay(self.sound)
```

为了创建新实例，我们就像调用函数那样调用类的名字。我们可以给对象定义新的实例变量，只要给它们赋个值就行了。于是，可以像下面这样创建一张幻灯片，并给它一幅图片和一段声音。

```
>>> slide1=slide()
```

```
>>> slide1.picture = makePicture(getMediaPath("barbara.jpg"))
>>> slide1.sound = makeSound(getMediaPath("bassoon-c4.wav"))
>>> slide1.show()
```

`slide.show()`函数显示图片并播放声音。`self`是什么东西？当执行`object.method()`时，Python会在对象所属的类中找到方法，并使用这个对象作为输入来调用它。将这个输入变量命名为`self`是Python的风格（因为它是对象自己）。有了保存在变量`self`中的对象，我们就可以通过`self.picture`和`self.sound`来访问它的图片和声音。

然而，如果必须在命令区设置所有变量，那用起来还是麻烦。能不能再简单一些呢？如果可以像真正的函数那样，把声音和图片作为类的输入传给幻灯片，情况会怎样呢？我们可以定义一种称为构造函数（constructor）的东西来达到这一目标。

为了通过一些输入来创建新实例，我们必须定义一个名为`__init__`的函数。怎么写？就是“下画线-下画线-i-n-i-t-下画线-下画线”。它是Python预定义的一个名字，用于命名初始化新对象的方法。我们的`__init__`方法需要三个输入：实例自身（因为所有方法都有这个参数）、一幅图像和一段声音。



程序149：幻灯片类

```
class slide:
    def __init__(self, pictureFile, soundFile):
        self.picture = makePicture(pictureFile)
        self.sound = makeSound(soundFile)

    def show(self):
        show(self.picture)
        blockingPlay(self.sound)
```

我们可以像下面这样使用`slide`类定义一张幻灯片。



程序150：使用我们的`slide`类放映幻灯片

```
def playSlideShow2():
    pictF = getMediaPath("barbara.jpg")
    soundF = getMediaPath("bassoon-c4.wav")
    slide1 = slide(pictF, soundF)
    pictF = getMediaPath("beach.jpg")
    soundF = getMediaPath("bassoon-e4.wav")
    slide2 = slide(pictF, soundF)
    pictF = getMediaPath("church.jpg")
    soundF = getMediaPath("bassoon-g4.wav")
    slide3 = slide(pictF, soundF)
    pictF = getMediaPath("jungle2.jpg")
    soundF = getMediaPath("bassoon-c4.wav")
    slide4 = slide(pictF, soundF)
    slide1.show()
    slide2.show()
    slide3.show()
    slide4.show()
```

使Python如此强大的特性之一就是Python中可以混合使用面向对象编程和函数式编程这两种风格。幻灯片现在成了对象，可以方便地保存在列表中，就像其他Python对象一样。下面是同样的幻灯片示例，这次我们用`map`来显示幻灯片。



程序151：对象中的幻灯片，函数式的放映

```
def showSlide(aSlide):
    aSlide.show()

def playSlideShow3():
    pictF = getMediaPath("barbara.jpg")
    soundF = getMediaPath("bassoon-c4.wav")
    slide1 = slide(pictF, soundF)
    pictF = getMediaPath("beach.jpg")
    soundF = getMediaPath("bassoon-e4.wav")
    slide2 = slide(pictF, soundF)
    pictF = getMediaPath("church.jpg")
    soundF = getMediaPath("bassoon-g4.wav")
    slide3 = slide(pictF, soundF)
    pictF = getMediaPath("jungle2.jpg")
    soundF = getMediaPath("bassoon-c4.wav")
    slide4 = slide(pictF, soundF)

    map(showSlide, [slide1, slide2, slide3, slide4])
```

面向对象版本的幻灯片程序编写起来是否更简单呢？重复代码显然少了。它体现了封装（encapsulation）特性，对象的数据和行为定义在一个地方，且只在一个地方，因此数据和行为中有一项发生改变就很容易体现在另一项上。支持使用大量对象的特性称为聚合（aggregation），这是一种非常强大的思想。我们不用反复定义新类——可以经常使用已知的强大结构，比如包含已有对象的列表，来实现强大功能。

16.4.1 Joe the Box

面向对象教学中最早使用的例子叫做*Joe the Box*，它是由Adele Goldberg和Alan Kay开发的。假如有如下的Box类：

```
class Box:
    def __init__(self):
        self.setDefaultColor()
        self.size=10
        self.position=(10,10)
    def setDefaultColor(self):
        self.color = red
    def draw(self, canvas):
        addRectFilled(canvas, self.position[0],
self.position[1], self.size, self.size, self.color)
```

那么，执行以下代码会产生什么结果呢？

```
>>> canvas = makeEmptyPicture(400,200)
>>> joe = Box()
>>> joe.draw(canvas)
>>> show(canvas)
```

我们把程序跟踪一遍：

- 显然，第一行程序只是创建了400像素宽、200像素高的一块白色画布（canvas）。
- 创建joe的时候，__init__方法得以调用。__init__从对象joe上调用了setDefaultColor，于是得到了默认的红色。执行到self.color = red时，实例变量color得以创建并获得

红色值。然后返回__init__，接着joe获得了尺寸10和位置 (10, 10) (size和position都成了新的实例变量)。

- 让joe在画布上画出自己的时候，他在x=10, y=10的位置把自己画成了一个红色的、各边长度都是10个像素的实心矩形 (addRectFilled)。
- 可以为Box类添加一个方法，让joe可以改变自己的尺寸。

```
class Box:
    def __init__(self):
        self.setDefaultColor()
        self.size=10
        self.position=(10,10)
    def setDefaultColor(self):
        self.color = red
    def draw(self,canvas):
        addRectFilled(canvas, self.position[0],self.
            position[1], self.size, self.size, self.color)
    def grow(self,size):
        self.size=self.size+size
```

现在，可以让joe增大或缩小。为grow输入一个负值，比如-2，会使joe缩小；正值则会使joe增大——当然，如果我们想让他增大很多但仍然完整地显示在画布上，还需要添加一个move方法。

现在，考虑把如下代码也加进程序区。

```
class SadBox(Box):
    def setDefaultColor(self):
        self.color=blue
```

注意，SadBox把Box列为超类（父类）。这意味着SadBox继承了Box的所有方法。如果执行以下代码，会出现什么结果呢？

```
>>> jane = SadBox()
>>> jane.draw(canvas)
>>> repaint(canvas)
```

我们再来跟踪这个程序：

- 创建SadBox的实例jane的时候，Box类中的__init__方法得以执行。
- __init__方法中发生的第一件事是从输入对象self上面调用了setDefaultColor。这个对象现在是jane，因此调用了jane的setDefaultColor。我们说SadBox的setDefaultColor重置 (override) 了Box的版本。
- jane的setDefaultColor把自己的颜色置为蓝色。
- 然后，返回Box的__init__方法，继续执行其他部分。把jane的尺寸设为10，位置设为(10, 10)。
- 让jane画出自己的时候，她显示为10×10的蓝色方块，出现在坐标为(10, 10)的位置。如果没有移动或增大joe，当jane画在他上面时，joe就消失了。

注意，joe和jane是不同种类的Box。他们拥有相同的实例变量（但实例变量的值不同），会做的事情也差不多。比如，他们都知道如何画出自己 (draw)，我们说draw是多态的 (polymorphic)。多态本来就是多种形式的意思。

SadBox (jane) 在创建的时候行为与Box稍有不同, 因此它知道一些不同的事情。joe和jane的例子体现了面向对象编程中的一些基本思想: 继承、在子类中特化 (specialization)、相同的实例变量和不同的实例变量值。

16.4.2 面向对象的媒体

当然, 我们一直在使用对象。图片、声音、样本和颜色, 这些都是对象。我们的像素列表和样本列表显然是聚合的例子。我们一直在使用的函数实际上只是掩盖了它内部使用的方法。当然可以直接调用对象的方法, 也可以使用常规的构造函数创建一幅图片。

```
>>> pic=Picture(getMediaPath("barbara.jpg"))
>>> pic.show()
```

函数show就是用下面的代码定义的。你可以忽略raise和__class__。关键问题是: 函数只是执行已有的图片方法show。

```
def show(picture):
    if not picture.__class__ == Picture:
        print "show(picture): Input is not a picture"
        raise ValueError
    picture.show()
```

有没有注意到? 我们为幻灯片定义了方法show(), 它与显示图片的方法名字一样。首先, 显然可以这么做——对象有自己的方法, 方法的名字可以与其他对象中的方法名相同。更强大的是: 每一个同名方法都可以实现相同的目标, 但实现的方式可以不同。对幻灯片和图片来说, show()方法做的都是“显示这个对象”。但不同的情形中真正发生的事情是不一样的: 图片只是显示了自己, 幻灯片则显示了图片, 同时播放了声音。

计算机科学思想: 多态

相同的名字可用来调用实现相同目标的不同方法, 我们把这一机制称为多态 (polymorphism)。对程序员来说, 这是一种非常强大的机制。只需让对象show()——不必关心方法到底执行了什么, 甚至不必知道正在show的这个对象到底是什么对象。作为程序员, 你只是指定自己的目标: 显示对象。面向对象的程序处理剩下的事情。

我们在JES中使用的方法包含许多多态的例子[⊖]。例如, 像素和颜色都懂得setRed、getRed、setBlue、getBlue、setGreen和getGreen这些方法。这样, 我们就不必单独取出像素中的颜色对象, 可以直接从像素上处理它们。我们可以定义接受两种输入的函数, 也可以针对每种输入提供不同的函数, 但这两种做法理解起来都不太清爽。还是用方法更方便。

```
>>> pic=Picture(getMediaPath("barbara.jpg"))
>>> pic.show()
>>> pixel = pic.getPixel(100,200)
>>> print pixel.getRed()
73
```

⊖ 别忘了JES是Jython的编程环境, Jython是一种特殊的Python。媒体支持是JES的一部分——它们不是Python核心的一部分。


```
>>> color = pixel.getColor()
>>> print color.getRed()
73
```

另一个例子是writeTo()方法。图片和声音都定义了writeTo(filename)方法。你有没有把writePictureTo()和writeSoundTo()搞混的经历？如果每次只写writeTo(filename)，是不是简单一些呢？这就是为什么这个方法在两个类中的名字是一样的，以及为什么多态如此强大。（你可能疑惑，为什么我们没有一开始就介绍这些呢？如果在第2章就讨论点号语法和多态方法，你觉得自己准备好了吗？）

整体来讲，在JES中，方法比函数多得多。再具体一点，JES中有一大堆用于在图片上绘图的功能，它们没法通过某个JES函数来使用。

- 可以想象，图片会懂得这样的一些方法：pic.addRect(color, x, y, width, height)、pic.addRectFilled(color, x, y, width, height)、pic.addOval(color, x, y, width, height)和pic.addOvalFilled(color, x, y, width, height)。

图16.11显示了矩形方法的示例，它是用下面的例子画出来的。

```
>>> pic=Picture (getMediaPath("640x480.jpg"))
>>> pic.addRectFilled (orange,10,10,100,100)
>>> pic.addRect (blue,200,200,50,50)
>>> pic.show()
>>> pic.writeTo("newrects.jpg")
```

图16.12给出了一些椭圆的示例，它是用下面的例子画出来的。

```
>>> pic=Picture (getMediaPath("640x480.jpg"))
>>> pic.addOval (green,200,200,50,50)
>>> pic.addOvalFilled (magenta,10,10,100,100)
>>> pic.show()
>>> pic.writeTo("ovals.jpg")
```

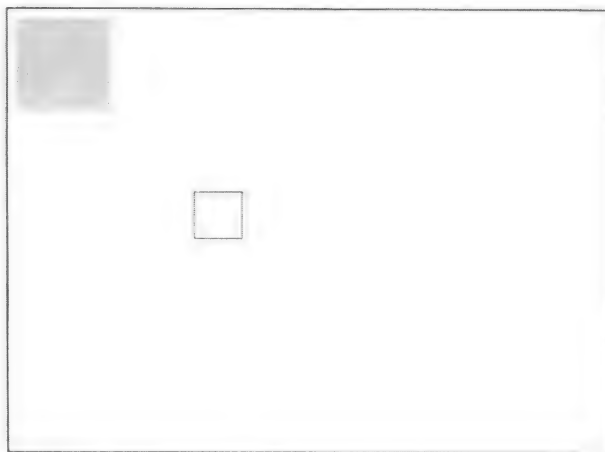


图16.11 矩形方法示例

- 图片还懂得如何绘制弧线。弧线就是圆的一部分。两种绘制弧线的方法分别是：pic.addArc(color, x, y, width, height, startAngle, arcAngle)和pic.addArcFilled(color, x, y, width, height, startAngle, arcAngle)。它们画出度数为arcAngle的弧线，起点为startAngle，0度对应钟表的3点方向。正度数对应逆

时针方向，负度数对应顺时针方向。圆（或椭圆）的中心就是 (x, y) 和 `width`、`height` 所定义的矩形的中心。

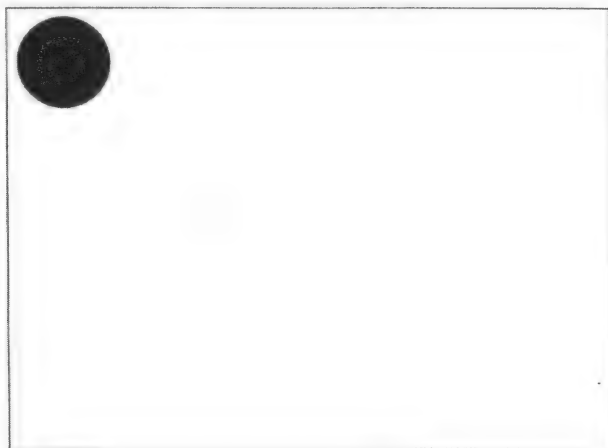


图16.12 椭圆方法示例

- 我们还可以用 `pic.addLine(color, x1, y1, x2, y2)` 画出彩色的直线。图16.13给出了弧线和直线的示例，它是用下面的例子画出来的。

```
>>> pic=Picture (getMediaPath("640x480.jpg"))
>>> pic.addArc(red,10,10,100,100,5,45)
>>> pic.show()
>>> pic.addArcFilled (green,200,100,200,100,1,90)
>>> pic.repaint()
>>> pic.addLine(blue,400,400,600,400)
>>> pic.repaint()
>>> pic.writeTo("arcs-lines.jpg")
```

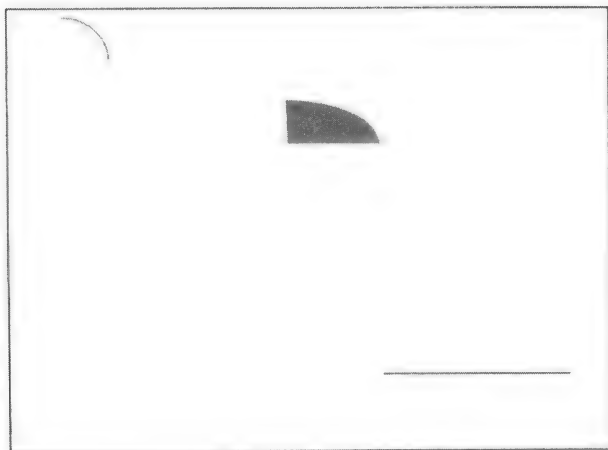


图16.13 圆弧方法示例

- Java中的文本可以有样式，但只限于所有平台都能显示的那些。 `pic.addText(color, x, y, string)` 是你能想到的。此外，还有 `pic.addTextWithStyle(color, x, y,`

string, style), 它接受一个样式 (style), 可通过makeStyle(font, emphasis, size)创建。其中, font可以是sansSerif、serif或mono, emphasis可以是italic、bold或plain, 或者把它们加起来得到的组合效果 (比如italic + bold)。size是以磅 (point) 为单位的字体大小。

图16.14给出了文本的示例, 它是用下面的例子画出来的。

```
>>> pic=Picture (getMediaPath("640x480.jpg"))
>>> pic.addText(red,10,100,"This is a red
string!")
>>> pic.addTextWithStyle (green,10,200,"This is a
bold, italic, green, large string",
makeStyle(sansSerif, bold+italic,18))
>>> pic.addTextWithStyle (blue,10,300,"This is a
blue, larger, italic-only, serif string",
makeStyle(serif, italic,24))
>>> pic.writeTo("text.jpg")
```

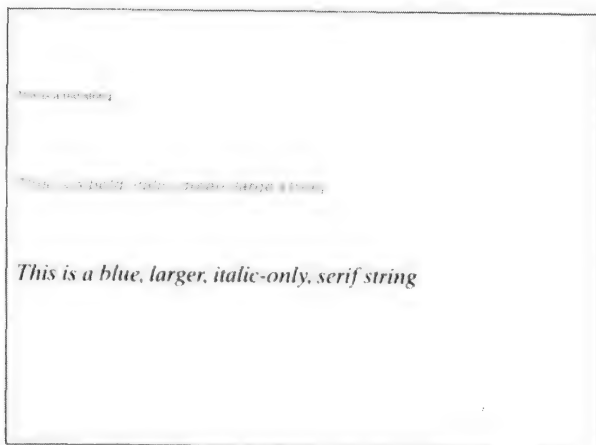


图16.14 文本方法示例

之前写过的媒体函数可以改写成方法的形式。我们需要创建Picture的子类, 然后为它添加方法。



程序152: 使用方法制作日落效果

```
class MyPicture(Picture):
    def makeSunset(self):
        for p in getPixels(self):
            p.setBlue(int(p.getBlue()*0.7))
            p.setGreen(int (p.getGreen()*0.7))
```

它可以这样使用:

```
>>> pict = MyPicture(getMediaPath("beach.jpg"))
>>> pict.explore()
>>> pict.makeSunset()
>>> pict.explore()
```

我们也可以创建Sound的子类和新方法来处理声音对象。访问声音样本值的方法是getSampleValue()和getSampleValueAt(index)。



程序153：使用方法做声音反序

```
class MySound(Sound):
    def reverse(self):
        target = Sound(self.getLength())
        sourceIndex = self.getLength() - 1
        for targetIndex in range(0, target.getLength()):
            sourceValue = self.getSampleValueAt(sourceIndex)
            target.setSampleValueAt(targetIndex, sourceValue)
            sourceIndex = sourceIndex - 1
        return target
```

它可以这样使用：

```
>>> sound = MySound(getMediaPath("always.wav"))
>>> sound.explore()
>>> target = sound.reverse()
>>> target.explore()
```

16.4.3 为什么使用对象

对象的作用之一是减少必须记住的名字数量。通过多态，你只需记住名字和目标，而不必记住所有不同的全局函数。

更重要的是：对象封装了数据和行为。想象一下，你想改变一个实例变量的名字，然后所有使用这个变量的方法都要改变。那样一来，需要改变的地方太多了，如果漏掉一个怎么办？能在一个地方全部改完是很有好处的。

聚合是对象系统的又一大好处。你可以有很多对象，它们都做着有用的事情。还需要更多？那就再创建一些。

Python的对象与许多语言中的对象类似。较大的一处不同是对实例变量的访问。在Python中，任何对象都可以访问并操作其他对象的实例变量。Java、C++和Smalltalk中却不是这样。在这些语言中，从其他对象上访问实例变量是受限的，有的甚至完全禁止——只能使用称为getter或setter（用于取得或设置实例变量）的方法来访问对象的实例变量。

对象系统的另一重要部分是继承（inheritance）。我们在小海龟和方块的例子中已经看到：可以把一个类（子类）声明为继承另一个类（父类）。继承提供了即时的多态——子类实例自动拥有父类的数据和行为，然后子类可以添加父类没有的其他数据和行为。这一机制可描述为子类成为父类的特化（specialization）。比如，只要声明class rectangle3D(rectangle)，就可以让一个三维长方体实例理解矩形实例所知的、能做的一切事情。

继承在面向对象的世界里是人们津津乐道的机制，然而，使用继承尚需权衡一二。它进一步减少了代码重复，这是好事。在实际编程中，继承却不像面向对象的其他有利机制（比如聚合和封装）用得那么多，而且它会让人困惑。输入下面的命令时，实际调用的是哪个类的方法呢？从命令本身很难看出来（draw方法是继承来的还是重置过的？——译者注），而且一旦出错，问题的位置也难以查找。

```
myBox = rectangle3D()
myBox.draw()
```

那什么时候使用对象呢？当你有成群成组的数据（比如图片和声音），而且想针对这一群或一组中的所有实例定义行为时，就应该定义自己的对象类了。你永远应该使用已有的对象，它们非常强大。如果你对点号语法或对象的思想感到不舒服，那就继续使用函数——函数也很好。对象只是在面对更加复杂的系统时会助你一臂之力。

编程摘要

以下是我们在这一章见过的一些程序片段：

面向对象编程

<code>class</code>	用于定义一个类。关键字 <code>class</code> 后面跟着一个类名，然后是可选的写在括号中的超类，最后以冒号结尾。跟在 <code>class</code> 语句下面的是方法，缩进定义在类的块中
<code>__init__</code>	对象初次创建时调用的方法。不是必须的

图形方法

<code>addRect, addRectFilled</code>	<code>Picture</code> 类中的方法，用于画矩形和实心矩形
<code>addOval, addOvalFilled</code>	<code>Picture</code> 类中的方法，用于画椭圆和实心椭圆
<code>addArc, addArcFilled</code>	<code>Picture</code> 类中的方法，用于画弧线和实心扇形
<code>addText, addTextWithStyle</code>	<code>Picture</code> 类中的方法，用于绘制文本和带有样式元素（比如粗体和sans serif 字体）的文本

<code>addLine</code>	<code>Picture</code> 类中的方法，用于画直线
<code>getRed, getGreen, getBlue</code>	<code>Pixel</code> 和 <code>Color</code> 对象上都可使用的方法，用于获取红色、绿色和蓝色分量
<code>setRed, setGreen, setBlue</code>	<code>Pixel</code> 和 <code>Color</code> 对象上都可使用的方法，用于设置红色、绿色和蓝色分量

习题

- 16.1 回答以下问题。
- 实例和类的区别是什么？
 - 函数和方法有哪些不同？
 - 面向对象编程和过程式编程有哪些不同？
 - 什么是多态？
 - 什么是封装？
 - 什么是聚合？
 - 什么是构造函数？
 - 生物细胞如何影响了对象思想的形成？
- 16.2 回答以下问题。
- 什么是继承？
 - 什么是超类？
 - 什么是子类？
 - 子类会继承父类的哪些方法？
 - 子类会继承父类的哪些实例变量（字段）？

- 16.3 为Turtle类添加一个画等边三角形的方法。
- 16.4 为Turtle类添加一个按指定宽度和高度画矩形的方法。
- 16.5 为Turtle类添加画一栋简单房子的方法。可以用矩形做房体，等边三角形做房顶。
- 16.6 为Turtle类添加画出一排房子和街道的方法。
- 16.7 为Turtle类添加画出一个字母的方法。
- 16.8 为Turtle类添加一个方法，画出你名字中的各个首字母。
- 16.9 制作一段电影，画面中有多只小海龟在移动。
- 16.10 为Slide类再添加一个构造函数，只接受一个图片文件名。
- 16.11 创建一个SlideShow类，持有一组幻灯片的列表，依次放映列表中的每张幻灯片。
- 16.12 创建一个CartoonPanel类，接受一个Picture数组，从左到右依次显示每张图片。它还应该有标题和作者信息，并将标题显示在左上方，作者显示在右上方。
- 16.13 创建一个Student类，每个学生应该有一个名字和一张照片，添加一个show方法显示学生的照片。
- 16.14 为SlideShow类添加一个字段来保存标题。修改show方法，首先放映一张只显示标题的图片。
- 16.15 创建一个PlayList类，接受一个声音列表，依次播放列表中的各段声音。
- 16.16 使用Picture类的方法画一个笑脸。
- 16.17 使用Picture类的方法画一道彩虹。
- 16.18 把图片镜像函数改写成MyPicture类中的方法。
- 16.19 修改Joe the Box示例：
 - 为Box类添加setColor方法，接受一种颜色作为输入，然后使输入的颜色成为方块的新颜色。（也许setDefaultColor应该调用setColor？）
 - 为Box类添加setSize方法，接受一个数字作为输入，然后使输入值成为方块的新尺寸。
 - 为Box类添加一个setPosition方法，接受一个列表或元组作为参数，然后使输入的坐标成为方块的新位置。
 - 修改__init__方法，让它使用setSize和setPosition，而不是直接设置实例变量。
- *16.20 完成Joe the Box示例：
 - (a) 实现grow和move。move方法接受一个相对距离作为输入，比如输入(-10, 15)则向上移动10个像素(x位置)，向右移动15个像素(y位置)。
 - (b) 创建joe和jane并通过它们画图案，移动一点儿，画一下，增大一点儿，再画一下，然后重绘整张画布。
- 16.21 制作一段电影，画面中有一些方块在增大或缩小。

深入学习

关于使用Python进行过程式、函数式和面向对象编程还有很多值得研究的内容。Mark推荐Mark Lutz（特别是[30]）和Richard Hightower[24]的书，说它们都是好书，可以把你引向Python编程中更深层的领域。你也可以看看Python网站（<http://www.python.org>）上的一些教程。

Python快速参考

A.1 变量

变量以字母开始，可以是除保留字（reserved word）以外的任何单词。保留字包括：and、assert、break、class、continue、def、del、elif、else、except、exec、finally、for、from、global、if、import、in、is、lambda、not、or、pass、print、raise、return、try、while、yield。

我们可以用print来显示表达式（比如一个变量）的值。如果只输入一个变量而不写print，那么就会得到变量的内部表示——函数或对象会给出它们在内存中的位置，字符串会带着引号显示出来。

```
>>> x = 10
>>> print x
10
>>> x
10
>>> y='string'
>>> print y
string
>>> y
'string'
>>> p=makePicture(pickAFile())
>>> print p
Picture, filename C:\ip-book\mediasources\
7inX95in.jpg height 684 width 504
>>> p
<media.Picture instance at 6436242>
>>> print sin(12)
-0.5365729180004349
>>> sin
<java function sin at 26510058>
```

A.2 函数创建

我们用def定义函数。def x(a, b):定义了一个名为“x”并接受两个输入值的函数，两个输入值将分别绑定到变量“a”和“b”上。函数体缩进跟在def语句之后。

函数可以用return语句来返回值。

A.3 循环和条件式

我们用for创建大多数循环，for语句接受一个索引变量和一个列表。循环体针对列表中的每个元素执行一次。

```
>>> for p in [1,2,3]:
...     print p
...
1
2
3
```

`for`中的列表常常使用`range`函数产生。`range`可接受一个、两个或三个输入。仅有一个输入时，`range`的范围从0开始到输入值之前为止。两个输入时，`range`的范围从第一个输入值开始，到第二个输入值之前结束。三个输入时，`range`的范围从第一个输入值开始，每次步进第三个输入指定的值，直到第二个输入值之前结束。

```
>>> range(4)
[0, 1, 2, 3]
>>> range(1,4)
[1, 2, 3]
>>> range(1,4,2)
[1, 3]
```

`while`循环接受一个逻辑表达式，只要逻辑表达式为真就一直执行后面的代码块。

```
>>> x = 1
>>> while x < 5:
...     print x
...     x = x + 1
...
1
2
3
4
```

`break`立即终止当前循环。

`if`语句接受一个逻辑表达式并对表达式求值。如果为真，`if`的语句块便会执行；如果为假，存在`else`:子句时，则执行`else`:的语句块。

```
>>> if a < b:
...     print "a is smaller"
... else:
...     print "b is smaller"
```

A.4 操作符和数据表示函数

<code>+, -, *, /, **</code>	加、减、乘、除和乘方。优先次序是代数
<code><, >, ==, <=, >=</code>	逻辑运算符小于、大于、等于、小于或等于、大于或等于
<code><>, !=</code>	逻辑运算符不等于（两种写法等价）
<code>and, or, not</code>	逻辑连词与、或、非
<code>int()</code>	返回输入值（浮点数或字符串）的整数部分
<code>float()</code>	返回输入值的浮点数版本
<code>str()</code>	返回输入值的字符串表示
<code>ord()</code>	输入一个字符，返回其ASCII数字表示

A.5 数值函数

<code>abs()</code>	绝对值
<code>sin()</code>	正弦
<code>cos()</code>	余弦
<code>max()</code>	所有输入值（包括列表）中的最大值
<code>min()</code>	所有输入值（包括列表）中的最小值
<code>len()</code>	返回输入序列的长度

A.6 序列操作

序列（字符串、列表、元组）可以相加，用于连接序列（如`s1 + s2`）。

序列中的元素可以用下标访问：

- `seq[n]`访问序列中的第 n 个元素（第一个元素下标为0）。
- `seq[n:m]`访问序列中从第 n 个元素开始，到第 m 个元素（但不包括 m ）之间的元素。
- `seq[:m]`访问序列中从开头到第 m 个元素（但不包括 m ）之间的元素。
- `seq[n:]`访问序列中从第 n 个元素到序列末尾之间的元素。

A.7 字符串转义

<code>\t</code>	跳格（Tab）
<code>\b</code>	退格
<code>\n</code>	换行
<code>\r</code>	回车
<code>\uXXXX</code>	Unicode字符，XXXX为字符的十六进制编码

在字符串之前加一个“`r`”，比如`r"C:\mediasources"`，可以忽略所有转义符，以原始模式（raw mode）处理字符串。

A.8 常用字符串方法

- `count(sub)`：返回子串`sub`在字符串中出现的次数。
- `find(sub)`：返回子串`sub`在字符串中出现的下标，如果字符串中找不到`sub`则返回-1。
`find`可接受指定起始位置和结束位置的可选参数。`rfind`接受同样的输入但从右向左查找字符串。
- `upper()`，`lower()`：分别将字符串转换成全部大写或全部小写。
- `isalpha()`，`isdigit()`：分别当字符串中的字符全部为字母或者全部为数字时返回真。
- `replace(s, r)`：将字符串中出现的所有“`s`”替换为“`r`”。
- `split(d)`：返回以字符`d`为分隔点拆分后的子串列表。

A.9 文件

文件通过`open`函数打开，`open`接受两个输入：文件名和文件模式。文件模式“`r`”表示读，“`w`”表示写，“`a`”表示追加，可以连上一个“`t`”表示文本或“`b`”表示二进制。文件方法包括：

- `read()`: 以字符串形式返回整个文件。
- `readlines()`: 以按行分隔的字符串列表形式返回整个文件。
- `write(s)`: 将字符串`s`写入文件。

A.10 列表

与序列类似，列表也使用中括号（`[]`）索引，使用加号（`+`）连接。列表方法包括：

- `append(a)`: 将项目`a`附加到列表中。
- `remove(b)`: 从列表中删除项目`b`。
- `sort()`: 列表排序。
- `reverse()`: 列表反转。
- `count(s)`: 返回元素`s`在列表中出现的次数。

A.11 字典、散列表和关联数组

字典使用大括号（`{}`）创建，使用键（`key`）访问。

```
>>> d = {'cat': 'Diana', 'dog': 'Fido'}
>>> print d
{'cat': 'Diana', 'dog': 'Fido'}
>>> print d.keys()
['cat', 'dog']
>>> print d['cat']
Diana
```

A.12 外部模块

模块使用`import`来访问，可以使用别名来输入，比如`import javax.swing as swing`。可以用`from module import n1, n2`这种形式导入特定代码，访问这些片段时不需要点号语法。还可以用`from module import *`导入模块中的所有代码，同样不需要使用点号语法访问。

A.13 类

类使用关键字`class`创建，`class`之后跟类的名字，然后还可以在括号中指定可选的（一个或多个）超类。方法缩进跟在`class`语句之后。构造函数（创建类的新实例时调用）必须命名为`__init__`。一个Python类可以有多个构造函数，只要接受不同的参数即可。

A.14 函数式方法

<code>apply</code>	接受一个函数和一个列表作为输入，其中列表中的元素个数与输入函数接受的参数个数一致，然后 <code>apply</code> 基于列表中的输入参数调用传入的函数
<code>map</code>	接受一个函数和一个包含多项输入的列表作为输入。针对每个列表元素调用传入的函数，并返回各次输出结果（返回值）组成的列表
<code>filter</code>	接受一个函数和一个包含多项输入的列表作为输入。针对每个列表元素调用传入的函数，如果输入函数返回真（非0），则返回的结果中将包含该元素
<code>reduce</code>	接受一个函数和一个包含多项输入的列表作为输入。输入函数首先应用到前两个列表元素上，返回的结果跟下一个列表元素一起作为下一次函数调用的输入，返回的结果再跟下一个列表元素一起作为再下一次函数调用的输入，以此类推。最后一次调用的结果作为 <code>reduce</code> 的结果返回

参考文献

1. AAUW, *Tech-Savvy: Educating Girls in the New Computer Age*, American Association of University Women Education Foundation, New York, 2000.
2. HAROLD ABELSON, GERALD JAY SUSSMAN, AND JULIE SUSSMAN, *Structure and Interpretation of Computer Programs*, 2d ed., MIT Press, Cambridge, MA, 1996.
3. KEN ABERNETHY AND TOM ALLEN, *Exploring the Digital Domain: An Introduction to Computing with Multimedia and Networking*, PWS Publishing, Boston, 1998.
4. ACM/IEEE, *Computing Curriculum 2001*, <http://www.acm.org/sigcse/cc2001> (2001).
5. BETH ADELSON AND ELLIOT SOLOWAY, "The Role of Domain Experience in Software Design," *IEEE Transactions on Software Engineering* SE-11 (1985), no. 11, 1351–1360.
6. JENS BENNEDSEN AND MICHAEL E. CASPERSEN, "Failure Rates in Introductory Programming," *SIGCSE Bulletin* 39 (2007), no. 2.
7. RICHARD BOULANGER (ed.), *The Csound Book: Perspectives in Synthesis, Sound Design, Signal Processing, and Programming*, MIT Press, Cambridge, MA, 2000.
8. AMY BRUCKMAN, "Situated Support for Learning: Storm's Weekend with Rachael," *Journal of the Learning Sciences* 9 (2000), no. 3, 329–372.
9. JOHN T. BRUER, *Schools for Thought: A Science of Learning in the Classroom*, MIT Press, Cambridge, MA, 1993.
10. ALAN J. DIX, JANET E. FINLAY, GREGORY D. ABOWD, AND RUSSELL BEALE, *Human-Computer Interaction*, 2d ed., Prentice Hall, Upper Saddle River, NJ, 1998.
11. CHARLES DODGE AND THOMAS A. JERSE, *Computer Music: Synthesis, Composition, and Performance*, Schirmer-Thomson Learning, New York, 1997.
12. MATTHIAS FELLEISEN, ROBERT BRUCE FINDLER, MATTHEW FLATT, AND SHRIRAM KRISHNAMURTHI, *How to Design Programs: An Introduction to Programming and Computing*, MIT Press, Cambridge, MA, 2001.
13. ANN E. FLEURY, "Encapsulation and Reuse as Viewed by Java Students," *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education* (2001), 189–193.
14. JAMES D. FOLEY, ANDRIES VAN DAM, AND STEVEN K. FEINER, *Introduction to Computer Graphics*, Addison Wesley, Reading, MA, 1993.
15. ANDREA FORTE AND MARK GUZDIAL, *Computers for Communication, Not Calculation: Media as a Motivation and Context for Learning*, HICSS 2004, Big Island, HI, IEEE Computer Society 2004.
16. DANNY GOODMAN, *JavaScript & DHTML Cookbook*, O'Reilly & Associates, Sebastapol, CA, 2003.
17. MARTIN GREENBERGER, "Computers and the World of the Future," transcribed recordings of lectures at the Sloan School of Business Administration, April 1961, MIT Press, Cambridge, MA, 1962.
18. RASHI GUPTA, *Making Use of Python*, Wiley, New York, 2002.
19. MARK GUZDIAL, *Squeak: Object-Oriented Design with Multimedia Applications*,

- Prentice Hall, Englewood, NJ, 2001.
20. MARK GUZDIAL AND KIM ROSE (eds.), *Squeak, Open Personal Computing for Multimedia*, Prentice Hall, Englewood, NJ, 2001.
 21. MARK GUZDIAL AND ALLISON ELLIOT TEW, "Imagineering Inauthentic Legitimate Peripheral Participation: An Instructional Design Approach for Motivating Computing Education," paper presented at the International Computing Education Research Workshop, Canterbury, UK, ACM, New York, NY, 2006.
 22. IDIT HAREL AND SEYMOUR PAPERT, "Software Design as a Learning Environment," *Interactive Learning Environments* 1 (1990), no. 1, 1–32.
 23. BRIAN HARVEY, *Computer Science Logo Style*, 2d ed., Vol. 1: *Symbolic Computing*, MIT Press, Cambridge, MA, 1997.
 24. RICHARD HIGHTOWER, *Python Programming with the Java Class Libraries*, Addison-Wesley, Reading, MA, 2003.
 25. DAN INGALLS, TED KAEHLER, JOHN MALONEY, SCOTT WALLACE, AND ALAN KAY, "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself," *OOPSLA'97 Conference Proceedings*, ACM, Atlanta, 1997, pp. 318–326.
 26. JANET KOLODNER, *Case-Based Reasoning*, Morgan Kaufmann, San Mateo, CA, 1993.
 27. HEATHER PERRY, LAUREN RICH, AND MARK GUZDIAL, *A CSI Course Designed to Address Interests of Women*, ACM: New York ACM SIGCSE Conference 2004, Norfolk, VA, 2004, pp. 190–194.
 28. MARGARET LIVINGSTONE, *Vision and Art: The Biology of Seeing*, Harry N. Abrams, New York, 2002.
 29. FREDERIK LUNDH, *Python Standard Library*, O'Reilly and Associates, Sebastopol, CA, 2001.
 30. MARK LUTZ AND DAVID ASCHER, *Learning Python*, O'Reilly & Associates, Sebastopol, CA, 2003.
 31. JANE MARGOLIS AND ALLAN FISHER, *Unlocking the Clubhouse: Women in Computing*, MIT Press, Cambridge, MA, 2002.
 32. DAN OLSEN, *Developing User Interfaces*, Morgan Kaufmann Publishers, San Mateo, CA, 1998.
 33. M. RESNICK, *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*, MIT Press, Cambridge, MA, 1997.
 34. JEANNETTE WING, "Computational Thinking," *Communications of the ACM* Vol-49 (2006), no. 3, 33–35.
 35. CURTIS ROADS, *The Computer Music Tutorial*, MIT Press, Cambridge, MA, 1996.
 36. ROBERT SLOAN AND PATRICK TROY "CS 0.5: A Better Approach to Introductory Computer Science for Majors," *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, ACM Press: New York, 2008, 271–275.
 37. ALLISON ELLIOT TEW, CHARLES FOWLER, AND MARK GUZDIAL, "Tracking an Innovation in Introductory CS Education from a Research University to a Two-Year College," *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, ACM press: New York, 2005, 416–420.

Python计算与编程实践 多媒体方法 (原书第2版)

Introduction to Computing and Programming in Python A Multimedia Approach Second Edition

计算机程序设计课程往往是枯燥乏味的,而本书根据教育理论,开发了一种新的教学方法,介绍如何通过多媒体编程来学习计算机程序设计,将趣味性和实用性融于枯燥的编程课程的教学当中。本书方法独特新颖,实例通俗易懂,可帮助读者快速入门并深入掌握编程技能,是一本理论联系实际的设计教程。

本书使用的编程语言是Python,这是因为Python强大实用(比如网站开发)、易于入门,计算机专业和非专业人士都可以学习。本书以Python数字多媒体编程为主线,从图像、声音、文本和电影这些学生已经熟知的内容开始,讲解它们的处理方法,其中穿插介绍计算机科学与程序设计的基础知识。在讲解知识点的时候也独具匠心,先介绍容易理解的基本概念,如赋值、顺序操作、迭代、条件式、函数定义等,逐步涉及抽象内容,如算法复杂度、程序效率、计算机组成、层次式分解、递归、面向对象等。本书还提供了大量例题和习题,方便教学。

作者简介

Mark Guzdial 是佐治亚理工学院计算机学院交互式计算专业的教授。他是ACM国际计算机教育研究系列研讨会的创立者之一,ACM教育委员会副主席,“Journal of the Learning Sciences”和“Communications of the ACM”编委会委员。Guzdial博士主要关注计算机教育方面的研究。他的第一本著作论述Squeak语言及其在教育中的应用。他是Swiki(Squeak Wiki)的早期开发者,Swiki是第一个专门用于学校的wiki。他出版了多本关于利用多媒体编程环境学习计算机编程的著作,影响了世界各地的计算机本科生教学。

Barbara Ericson 是佐治亚理工学院计算机学院“计算机普及”课程的主管和研究人员。她从2004年开始就致力于改善计算机基础教育,现在是计算机科学教师协会的师范教育代表,美国女性信息技术中心K-12联盟的合作主席,计算机科学AP考试开发委员会成员。她的研究兴趣涉及计算机图形学、人工智能和面向对象编程等多个领域。

客服热线: (010) 88378991, 88361066
购书热线: (010) 68326294, 88379649, 68995259
投稿热线: (010) 88379604
读者信箱: hzsj@hzbook.com

PEARSON

www.pearson.com

华章网站 <http://www.hzbook.com>

网上购书: www.china-pub.com

封面设计: 包昂 林彦

上架指导: 计算机 程序设计

ISBN 978-7-111-38738-1



9 787111 387381

定价: 69.00元